Aprendiendo Korn Shell

Segunda Edición

Autores
Arnold Robbins
Bill Rosenblatt

Traducción
Omar Sánchez Enríquez

Diciembre 2023

COPYRIGHT

Copyright © 2002, 1993 O'Reilly & Associates, Inc. Todos los derechos reservados.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly & Associates pueden adquirirse para uso educativo, comercial o promocional. También están disponibles ediciones en línea para la mayoría de los títulos (safari.oreilly.com). Para más información, póngase en contacto con nuestro departamento de ventas corporativas/institucionales: (800) 998-9938 o *corporate@oreilly.com*.

Nutshell Handbook, el logotipo de Nutshell Handbook y el logotipo de O'Reilly son marcas registradas de O'Reilly & Associates, Inc. Muchas de las denominaciones utilizadas por los fabricantes y vendedores para distinguir sus productos son reclamadas como marcas comerciales. En los casos en que esas denominaciones aparecen en este libro, y O'Reilly & Associates, Inc. tenía conocimiento de una reclamación de marca, las denominaciones se han impreso en mayúsculas o con iniciales. La asociación entre la imagen de una tortuga de carey y el aprendizaje del caparazón de Korn es una marca comercial de O'Reilly & Associates, Inc.

Aunque se han tomado todas las precauciones en la preparación de este libro, el editor y los autores no asumen ninguna responsabilidad por los errores u omisiones, ni por los daños resultantes del uso de la información aquí contenida.

DEDICATORIA

A Margot Lorraine Rosenblatt, nacida el 16 de abril de 2001.

–Bill Rosenblatt

A mi esposa, Miriam, por su amor y apoyo. A mis hijos, Chana, Rivka, Nachum y Malka.

–Arnold Robbins

PREFACIO

La larga y tortuosa historia del sistema operativo Unix ha dado lugar a sistemas con todo tipo de permutaciones y combinaciones de características. Esto significa que cada vez que te acercas a un sistema Unix desconocido, necesitas averiguar ciertas cosas sobre él para poder utilizarlo correctamente. E incluso en un sistema dado, puedes tener un número de opciones que puedes hacer sobre las características que quieres usar.

La decisión más importante, si es que la tienes que tomar, es qué shell usar. «Shell» es la jerga de Unix para el programa que le permite comunicarse con el ordenador introduciendo comandos y obteniendo respuestas. El shell está completamente separado del sistema operativo Unix per se; es sólo un programa que se ejecuta en Unix. En otros sistemas como MS-DOS, Microsoft Windows, Macintosh,[1] OpenVMS y VM/CMS, el intérprete de comandos o la interfaz de usuario es una parte integral del sistema operativo.

Hoy en día hay docenas de shells diferentes flotando por ahí, que van desde el estándar original, el shell de Bourne, hasta interfaces basadas en menús y gráficas. Los shells más importantes han sido el shell de Bourne, el shell C y el shell de Korn, el tema de este libro.

Versiones de Korn Shell

En concreto, este libro describe la versión de 1993 del shell de Korn. La versión de 1993 se distribuye con las tres principales versiones comerciales de Unix: Solaris, HP-UX y AIX, aunque como parte del Entorno Común de Escritorio (CDE), en `/usr/dt/bin/dtksh`. La versión de 1988 también está disponible en los sistemas Unix modernos, normalmente como `/usr/bin/ksh`. Existen otras versiones, variaciones e implementaciones en otros sistemas operativos; éstas, junto con las diferencias entre las versiones de 1988 y 1993 del shell de Korn se describen en Apéndice A.

La versión de 1993 está ahora disponible tanto en forma de código fuente como de ejecuta-

[1] Al menos hasta MacOS X. MacOS X tiene un sistema similar a Unix en su núcleo: Darwin, que deriva de Mach y 4.4-BSD-Lite.

bles precompilados para muchos sistemas comunes. En Apéndice C se describe la descarga y construcción de la misma. La última versión descargable de *ksh* tiene una serie de características que no estaban en las versiones anteriores. También cubrimos todas esas características. Hemos hecho un esfuerzo para señalar cuando algo puede no estar en una versión anterior, pero caveat emptor; si necesita una característica y la versión de su sistema del shell de Korn 1993 no la tiene, puede necesitar descargar un ejecutable precompilado o descargar el código fuente y construir su propio ejecutable.

Para saber qué versión tienes, escribe el comando *set -o emacs*, y luego pulsa CTRL-V. Deberías ver una fecha seguida de la letra de la versión (la letra no suele ser importante). Si es así, tienes una de las versiones oficiales, la de 1993, la de 1988 o una más antigua. Pero si no es así, tiene una versión no estándar, como pdksh, el shell de Korn de dominio público que se discute en el Apéndice A.

Resumen de las características del Shell de Korn

El shell de Korn es el más avanzado de los shells que se distribuyen «oficialmente» con los sistemas Unix. Se trata de un sucesor evolutivo del shell de Bourne, compatible con las versiones anteriores, que incluye la mayoría de las principales ventajas del shell C, así como una serie de nuevas características propias.

Entre las características apropiadas del intérprete de comandos C se incluyen:

Control de trabajo: La capacidad de detener trabajos con CTRL-Z y moverlos al primer plano o al fondo con los comandos *fg* y *bg*.

Alias: La capacidad de definir nombres abreviados para comandos o líneas de comando.

Funciones: La capacidad de almacenar su propio código de shell en la memoria en lugar de archivos. Las funciones aumentan la programabilidad y la eficiencia. (Las funciones han sido comunes en el shell de Bourne durante muchos años).

Historial de comandos: La capacidad de recordar comandos introducidos previamente.

Las principales características nuevas del Shell de Korn incluyen:

Edición de línea de comandos: Esta característica le permite usar comandos de edición de estilo *vi* o *Emacs* en sus líneas de comando.

Funciones de programación integradas: La funcionalidad de varios comandos exter-

nos de Unix, incluidos *test , expr , getopt* y *echo*, se ha integrado en el propio shell, lo que permite que las tareas de programación comunes se realicen de manera más limpia y sin crear procesos adicionales.

Estructuras de Control: Las estructuras adicionales de control de flujo, especialmente la construcción de selección , permiten una fácil generación de menús.

Primitivos de depuración: Estas características hacen posible escribir herramientas que ayuden a los programadores a depurar su código de shell.

Expresiones regulares: Bien conocidas por los usuarios de utilidades de Unix como *grep* y *awk*, las expresiones regulares (aunque con una sintaxis diferente) se han agregado al conjunto estándar de comodines de nombre de archivo y a la función de variable de shell.

Funciones de E/S avanzadas: Varias instalaciones nuevas para el control de E/S de procesos, incluida la capacidad de realizar una comunicación bidireccional con procesos concurrentes (corrutinas) y conectarse a servicios de red.

Nuevas opciones y variables: Estas opciones y variables le brindan más formas de personalizar su entorno que los shells estándar de Unix.

Mayor velocidad: El shell de Korn a menudo ejecuta el mismo programa de shell considerablemente más rápido que el shell de Bourne.

Características de seguridad: Funciones diseñadas para ayudar a proteger contra «caballos de Troya» y otros tipos de esquemas de robo.

Las principales características nuevas de la versión de 1993 incluyen:

Conformidad con POSIX: El cumplimiento de POSIX, un estándar internacional para la programación de shell portátil, hace posible escribir y utilizar scripts de shell portátiles.

Aritmética para bucles: Esta nueva estructura de control le permite programar de forma más natural cuando se repite un número fijo de veces.

Aritmética de punto flotante: La capacidad de usar números de coma flotante y las nuevas funciones aritméticas integradas enriquecen el shell como lenguaje de programación.

Nombres de variables estructuradas: La nueva sintaxis para los nombres de variables proporciona funciones similares a las estructuras C y los registros Ada para agrupar elementos relacionados en una variable.

Referencias a variables indirectas: Esta facilidad facilita la programación de funciones de shell para manipular variables globales.

matrices asociativas: Una potente función de gestión de datos que es similar a las de *awk* o *perl*.

Facilidades adicionales de manipulación de texto: Incluso hay más formas de hacer coincidir patrones y sustituir variables.

Más comandos integrados: Los comandos adicionales mejoran la eficiencia y aumentan la portabilidad de los scripts.

Público Objetivo

Este libro está diseñado para atraer a los usuarios ocasionales de Unix que están por encima del nivel de «principiante». Deberías estar familiarizado con el proceso de iniciar sesión, introducir comandos y hacer cosas sencillas con los archivos. Aunque el Capítulo 1 repasa conceptos como el esquema de archivos y directorios en forma de árbol, puede que te parezca que avanza demasiado rápido si eres un completo neófito. En ese caso, le recomendamos el libro *Learning the Unix Operating System*, de Jerry Peek, Grace Todino y John Strang, publicado por O'Reilly & Associates, Inc.

Si eres un usuario experimentado, puedes saltarte el Capítulo 1. Pero si su experiencia es con el shell C, puede encontrar que el Capítulo 1 revela algunas diferencias sutiles entre los shells Korn y C.

No importa cuál sea su nivel de experiencia, indudablemente aprenderá muchas cosas en este libro que le harán un usuario más productivo del shell de Korn – desde las características más importantes hasta los detalles a nivel de «novato» de los que no era consciente.

Si está interesado en la programación del shell (escribiendo scripts y funciones del shell que automatizan las tareas cotidianas o sirven como utilidades del sistema), también debería encontrar este libro útil. Sin embargo, hemos evitado deliberadamente hacer una fuerte distinción entre el uso interactivo del shell (introducir comandos durante una sesión de inicio de sesión) y la programación del shell. Consideramos que la programación del shell es una consecuencia natural e inevitable del aumento de la experiencia como usuario.

En consecuencia, cada capítulo depende de los anteriores, y aunque los tres primeros capítulos están orientados únicamente al uso interactivo, los siguientes describen características interactivas orientadas al usuario además de conceptos de programación.

De hecho, si este libro tiene un mensaje primordial, es éste: «El shell de Korn es un entorno de programación Unix increíblemente potente y muy infravalorado. Usted -sí, usted- puede escribir programas de shell útiles, incluso si acaba de aprender a conectarse la semana pasada y nunca ha programado antes».

Con este fin, hemos decidido no dedicar mucho tiempo a las características que interesan exclusivamente a los programadores de sistemas de bajo nivel. Conceptos como descriptores de archivos, tipos de archivos especiales, etc., sólo pueden confundir al usuario casual, y de todos modos, pensamos que aquellos de ustedes que entienden esas cosas son lo suficientemente inteligentes como para extrapolar la información necesaria de nuestras discusiones superficiales.

Ejemplos de Código

Este libro está lleno de ejemplos de comandos y programas del shell que están diseñados para ser útiles en tu vida diaria como usuario, no sólo para ilustrar la función que se explica. A partir del Capítulo 4, incluimos varios problemas de programación, que llamamos tareas, que ilustran conceptos particulares de programación del shell. Algunas tareas tienen soluciones que se perfeccionan en capítulos posteriores. Los capítulos posteriores también incluyen ejercicios de programación, muchos de los cuales se basan en las tareas del capítulo.

Puedes usar cualquier código que veas en este libro y pasarlo a tus amigos y colegas. Le animamos especialmente a que lo modifique y mejore usted mismo. El código está disponible en el sitio web de este libro: http://www.oreilly.com/catalog/korn2/.

Si desea probar ejemplos pero no usa el shell de Korn como su shell de inicio de sesión, debe colocar la siguiente línea en la parte superior de cada script de shell:

```
#!/bin/ksh
```

Si su shell de Korn no está instalado como el archivo */bin/ksh*, o si */bin/ksh* es la versión de 1988, sustituya el nombre de ruta completo por su versión de *ksh93* en lo anterior.

Resumen del Capítulo

Si desea investigar temas específicos en lugar de leer todo el libro, aquí hay un resumen capítulo por capítulo:

1. Capítulo 1

   Presenta el shell de Korn y le indica cómo instalarlo como su shell de inicio de sesión. Luego presenta los conceptos básicos del uso de shell interactivo, incluidas las descripciones generales del esquema de archivos y directorios de Unix, E/S estándar y trabajos en segundo plano.

2. Capítulo 2

   Describe el mecanismo de historial de comandos del shell, incluidos los modos de edición de *emacs* y *vi* y el comando *hist* history.

3. Capítulo 3

   Cubre formas de personalizar su entorno de shell sin programación, mediante el uso de los archivos *.profile* y de entorno. Los alias, las opciones y las variables de shell son las técnicas de personalización discutidas.

4. Capítulo 4

   Introduce la programación de shell. Este capítulo explica los conceptos básicos de las funciones y los scripts de shell, y analiza varias funciones de programación importantes: operadores de manipulación de cadenas, expresiones regulares, argumentos de línea de comandos (parámetros posicionales) y sustitución de comandos.

5. Capítulo 5

   Continúa la discusión de la programación del shell describiendo el estado de salida del comando, las expresiones condicionales y las estructuras de control de flujo del shell: *if* , *for* , *case* , *select* , *while* y *till*.

6. Capítulo 6

   Profundiza en los parámetros posicionales y el procesamiento de opciones de la línea de comandos, luego analiza los tipos especiales y las propiedades de las variables, como la aritmética de enteros y de coma flotante, la versión aritmética del bucle *for*, las matrices indexadas y asociativas y el comando *typeset*.

7. Capítulo 7

   Proporciona una descripción detallada de la E/S del shell de Korn, completando la

información omitida en el Capítulo 1 . Todos los redirectores de E/S del shell están cubiertos, junto con la capacidad del shell para realizar conexiones de socket TCP/IP y los comandos de E/S de línea a la vez *read , print* e *printf*. Luego, el capítulo analiza el mecanismo de procesamiento de la línea de comandos del shell y el comando *eval*.

8. Capítulo 8

   Cubre los problemas relacionados con el proceso en detalle. Comienza con una discusión sobre el control del trabajo y luego pasa a información de bajo nivel sobre los procesos, incluidos los ID de procesos, las señales y las trampas. Luego, el capítulo pasa a un nivel más alto de abstracción para analizar corrutinas, conductos bidireccionales y subcapas.

9. Capítulo 9

   Describe varias técnicas de depuración, comenzando con las simples como los modos de rastreo y detallado y las trampas de «señal falsa». A continuación, este capítulo describe las funciones de disciplina. Finalmente, presenta *kshdb*, una herramienta de depuración de shell de Korn que puede usar para depurar su propio código.

10. Capítulo 10

    Brinda información para administradores de sistemas, incluidas técnicas para implementar la personalización de shell en todo el sistema, personalizar los editores integrados y funciones relacionadas con la seguridad del sistema.

11. Apéndice A

    Compara el shell de Korn de 1993 con varios shells similares, incluido el shell SVR4 Bourne estándar, el shell de Korn de 1988, el shell estándar IEEE 1003.2 POSIX, el shell CDE Desk Top Korn (*dtksh*), *tksh* (que combina Tcl/Tk con *ksh*), el shell de Korn de dominio público (|emphpdksh), el |emphbash de la Free Software Foundation , el shell Z (|emphzsh) y varios shells de estilo Bourne (realmente entornos de emulación de Unix) para Microsoft Windows.

12. Apéndice B

    Contiene listas de opciones de invocación de shell, comandos integrados, alias predefinidos, variables integradas, operadores de prueba condicional, opciones de *comando de configuración, opciones de comando de composición tipográfica* y comandos de modo de edición de emacs y vi. Este apéndice también cubre los detalles completos para usar el comando integrado *getopts*.

13. Apéndice C

    Describe cómo descargar el código fuente de *ksh93* y crear un ejecutable que funcione. Este apéndice también cubre la descarga de ejecutables preconstruidos para varios sistemas diferentes.

14. Apéndice D

    Presenta los términos de licencia para el código fuente *ksh93*.

Convenciones Utilizadas en este Manual

Dejamos entendido que, cuando se introduce un comando del shell, se pulsa ENTER al final. ENTER está etiquetado como RETURN en algunos teclados.

Los caracteres llamados CTRL-X, donde X es cualquier letra, se introducen manteniendo pulsada la tecla CTRL (o CTL, o CONTROL) y pulsando esa letra. Aunque demos la letra en mayúscula, se puede pulsar la letra sin la tecla SHIFT.

Otros caracteres especiales son la nueva línea (que es lo mismo que CTRL-J), RETROCESO (lo mismo que CTRL-H), ESC, TAB y DEL (a veces etiquetado como DELETE o RUBOUT).

Este libro utiliza las siguientes convenciones tipográficas:

- *Cursiva*

  Se utiliza cuando se habla de nombres de archivos Unix, comandos externos e incorporados, nombres de alias, opciones de comandos, opciones del shell y funciones del shell. La cursiva también se utiliza en el texto cuando se discuten parámetros ficticios que deben ser reemplazados por un valor real, para distinguir los programas *vi* y *emacs* de sus modos Korn-shell, y para resaltar términos especiales la primera vez que se definen.

- `Anchura constante`

  Se utiliza para los nombres de las variables y las palabras clave del shell, los sufijos de los nombres de archivo y en los ejemplos para mostrar el contenido de los archivos o la salida de los comandos, así como para las líneas de comando cuando están dentro de un texto normal.

- **`Negrita de ancho constante`**

  Se utiliza en los ejemplos para mostrar la interacción entre el usuario y el shell;

cualquier texto que el usuario escriba se muestra en **negrita de ancho constante**. Por ejemplo:

```
$  pwd
/home/billr/ora/kb
$
```

- *Cursiva de ancho constante*

  Se utiliza en el texto y en las líneas de comando de ejemplo para los parámetros ficticios que deben sustituirse por un valor real. Por ejemplo:

  ```
  $  cd directorio
  ```

- *Vídeo inverso*

  Se utiliza en el Capítulo 2 para mostrar la posición del cursor en la línea de comandos que se está editando. Por ejemplo:

  ```
  grep -l Bob < ~pete/wk/nombres
  ```

- **NOTA** Indica un consejo, sugerencia o nota general.
- **ADVERTENCIA** Indica una advertencia o precaución.

Los comandos de utilidad estándar de Unix se mencionan a veces con un número entre paréntesis (normalmente 1) después del nombre del comando. El número se refiere a la sección del Manual del Usuario de Unix en la que encontrará la documentación de referencia (también conocida como «la página man») sobre la utilidad en cuestión. Por ejemplo, *grep*(1) significa la página del manual para *grep* en la Sección 1.

Cuando hay una diferencia importante entre las versiones de 1988 y 1993 del shell de Korn, nos referimos a ellas como *ksh88* y *ksh93* respectivamente. La mayor parte de este libro se aplica a todas las versiones del shell de Korn de 1993. Cuando necesitemos distinguir entre diferentes versiones del shell de Korn de 1993, añadiremos la versión menor al nombre, como *ksh93h*, o *ksh93l+*.

Acerca de la Segunda Edición

La primera edición de este libro cubría la versión de 1988 del shell de Korn. Poco después de su publicación, David Korn lanzó la versión de 1993, que incluía compatibilidad con el estándar de shell POSIX 1003.2, así como muchas nuevas características.

Aunque *ksh93* ha tardado en difundirse en el mundo comercial de Unix, el código fuente está ahora disponible, por lo que cualquiera que quiera una copia de la última y mejor versión de *ksh93* no tiene más que descargar el código fuente y compilarlo. Teniendo esto en cuenta, hemos hecho de *ksh93* el centro de la segunda edición, con un resumen de las diferencias disponible en un apéndice. Esta edición cubre la versión más reciente de *ksh93* disponible en el momento de escribir este artículo, que incluye algunas características significativas que no se encuentran en las versiones anteriores.

La estructura básica y el flujo del libro siguen siendo los mismos, aunque hemos corregido varios errores y erratas de la primera edición y hemos actualizado *kshdb*, el depurador de Korn Shell, para que funcione con *ksh93*. El Apéndice A incluye ahora más información sobre los programas similares al shell de Korn, tanto para sistemas Unix como Windows.

También se incluye en esta edición una tarjeta de referencia que cubre muchas de las características de *ksh93* descritas en este libro. Esta tarjeta tiene el copyright de Specialized Systems Consultants, Inc. (SSC), y se reimprime con permiso. SSC vende una versión de la tarjeta en cuatro colores, 26 paneles, de 3,5 por 8,5 pulgadas, que cubre tanto *ksh88* como *ksh93* con mucho más detalle. Para más información, consulte http://www.ssc.com.

Nos Gustaría Saber de Usted

Por favor, dirija sus comentarios y preguntas sobre este libro al editor:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (en Estados Unidos o Canadá)
(707) 829-0515 (internacional o local)
(707) 829-0104 (fax)

Disponemos de una página web para este libro, en la que aparecen ejemplos, erratas o cualquier información adicional. Puede acceder a esta página en:

http://www.oreilly.com/catalog/korn2/

Para comentar o hacer preguntas técnicas sobre este libro, envíe un correo electrónico a:

bookquestions@oreilly.com

Para obtener más información sobre nuestros libros, conferencias, centros de recursos y la red O'Reilly, consulte nuestro sitio web en:

http://www.oreilly.com

Agradecimientos

Escribir un libro desde cero no es fácil. Actualizar un libro es aún más difícil; el truco consiste en hacer imposible (o al menos difícil) que el lector sepa qué autor escribió cada parte. Espero haberlo conseguido. Quiero dar las gracias a Bill Rosenblatt por haber escrito la primera edición y haberme proporcionado un excelente material con el que trabajar. Es uno de los mejores libros de O'Reilly que he leído, y ha sido un placer trabajar con él.

Me gustaría dar las gracias (en orden alfabético) a Nelson A. Beebe (Departamento de Matemáticas de la Universidad de Utah), al Dr. David G. Korn (AT&T Research), a Chet Ramey (mantenedor de *bash*), a Bill Rosenblatt (GiantSteps/Media Technology Strategies) y al Dr. Eugene H. Spafford (Departamento de Ciencias de la Computación de la Universidad de Purdue) por haber revisado el libro y haber aportado muchos comentarios útiles. Mike Loukides, el editor del libro, fue muy paciente conmigo durante varios retrasos en la actualización. David Chu, del equipo editorial de O'Reilly, hizo un gran trabajo asegurándose de que muchas de las partes «tuercas y tornillos» del proyecto se hicieran, por lo que estoy agradecido.

David Korn, ahora de AT&T Research Laboratories, y autor del shell de Korn, respondió a varias preguntas y proporcionó acceso temprano a la documentación de *ksh93l*, lo que ayudó considerablemente, así como acceso previo a la liberación de *ksh93n*. Glenn Fowler, también de AT&T Research, me ayudó con problemas de compilación bajo GNU/Linux, así como a entender algunos de los puntos más finos del uso de *ksh*. Steve Alston aportó algunas mejoras al depurador *kshdb* en el Capítulo 9. George Kraft IV proporcionó información útil sobre *dtksh* para el Apéndice A. Glenn Barry, de Sun Microsystems, proporcionó información sobre zsh para el Apéndice A.

Gracias a Phil Hughes, presidente de SSC, por el permiso para reimprimir partes de su tarjeta de referencia de *ksh*.

Otros miembros del personal de O'Reilly también han contribuido al libro: Leanne Soylemez fue la editora de producción y correctora; Mary Brady y Jane Ellin proporcionaron un control de calidad adicional; Brenda Miller escribió el índice.

Por último, gracias a mi maravillosa esposa Miriam, por no reclamar la atención que le correspondía en demasiadas noches mientras trabajaba en este libro. Sin ella, no valdría la pena hacer nada.

Arnold Robbins
Nof Ayalon
ISRAEL

Agradecimientos de la Primera Edición

Muchas personas han contribuido a este libro de muchas maneras. Me gustaría dar las gracias a las siguientes personas por su asesoramiento y asistencia técnica: por la ayuda en la administración del sistema, John van Vlaanderen y Alexis Rosen. Por la información sobre shells alternativas, John (de nuevo), Sean Wilson (de MKS), Ed Ravin, Mel Rappaport y Chet Ramey. Por identificar la necesidad de un depurador de shell, la experiencia en SunOS y la seguridad del sistema y, de hecho, una parte importante de mi carrera, Hal Stern. Por las sugerencias de depuración, Tan Bronson. Por la ayuda humanitaria, Jessica Lustig. Y muchas gracias a David Korn por todo tipo de información de «boca de caballo» - y, por supuesto, por el propio shell de Korn.

Gracias a nuestros revisores técnicos: Jim Baumbach, Jim Falk, David Korn, Ed Miner, Eric Pearce y Ed Ravin. Aprecio especialmente la cooperación de Ed y Ed (en ese orden) durante mi fase de «¡¿Qué quieres decir con que no funciona?!».

Varias personas de O'Reilly & Associates contribuyeron a este esfuerzo: Gigi Estabrook y Clairemarie Fisher O'Leary corrigieron múltiples borradores del manuscrito, Kismet McDonough y Donna Woonteiler corrigieron el manuscrito, Len Muellner implementó el macropaquete de diseño del libro, Jennifer Niederst diseñó la portada y el formato del libro, y Chris Reilley creó las figuras. Por último, un océano de gratitud para Mike Loukides: editor, motivador, facilitador, constructivo y constante voz de la razón. Él y el resto de la gente de O'Reilly & Associates son algunas de las personas más innovadoras, interesantes y motivadas con las que he tenido el privilegio de trabajar.

–Bill Rosenblatt

Contenido

1. CONCEPTOS BÁSICOS DE KORN SHELL **1**

1.1. ¿Qué es un Shell? 1

1.2. Alcance de Este Libro 2

1.3. Historia de los Shells de Unix 3

1.3.1. El Shell de Korn 4

1.3.2. Características del Shell Korn 5

1.4. Cómo Obtener el Shell Korn de 1993 6

1.5. Uso Interactivo del Shell 7

1.5.1. Comandos, Argumentos y Opciones 8

1.5.2. Ayuda Incorporada (Built-in) 8

1.6. Ficheros 9

1.6.1. Directorios 10

1.6.2. Nombres de Archivo y Comodines 15

1.7. Entrada y Salida 19

1.7.1. E/S Estándar 20

1.7.2. Redirección de E/S 21

1.7.3. Tuberías (pipes) 23

1.8. Trabajos en Segundo Plano 24

1.8.1. E/S en Segundo Plano 26

1.8.2. Trabajos en Segundo Plano y Prioridades 27

1.9. Caracteres Especiales y Citas 28

1.9.1. Citando 29

1.9.2. Barra invertida 30

1.9.3. Comillas dobles 31

1.9.4. Continuación de líneas 32

1.9.5. Teclas de control 33

2. EDICIÓN DESDE LA LÍNEA DE COMANDOS **36**

2.1. Activando la Edición de Línea de Comandos . . . 37
2.2. El Archivo Histórico . . . 39
2.3. Modo de Edición Emacs . . . 40
2.3.1. Comando Básicos . . . 40
2.3.2. Comandos de palabras . . . 42
2.3.3. Comandos de línea . . . 43
2.3.4. Desplazarse por el archivo histórico . . . 43
2.3.5. Nombre de archivo y finalización y expansión de variable . . . 45
2.3.6. Comandos misceláneos . . . 47
2.3.7. Expansión de Macros con Alias . . . 50
2.4. Modo de Edición Vi . . . 51
2.4.1. Comandos de Modo de Control Simple . . . 52
2.4.2. Introducción y cambio de texto . . . 54
2.4.3. Comandos de eliminación . . . 55
2.4.4. Desplazarse por el archivo histórico . . . 58
2.4.5. Comandos de Búsqueda de Caracteres . . . 60
2.4.6. Nombre de Archivo y Finalización y Expansión de Variables . . . 61
2.4.7. Comandos Misceláneos . . . 63
2.4.8. Expansión de Macros con Alias . . . 64
2.5. El comando histórico . . . 64
2.6. Hábitos de los Dedos . . . 67

3. PERSONALIZACIÓN DEL ENTORNO **69**

3.1. El Archivo .profile . . . 70
3.1.1. El Archivo `/etc/profile` . . . 71
3.2. Alias . . . 72
3.3. Opciones . . . 76
3.4. Variables de Shell . . . 77
3.4.1. Variables y Citas . . . 79
3.4.2. Variables Incorporadas . . . 80
3.5. Personalización y Subprocesos . . . 92
3.5.1. Variables de Entorno . . . 93
3.5.2. El Archivo de Entorno . . . 94
3.6. Sugerencias de Personalización . . . 96

4. PROGRAMACIÓN BÁSICA DE SHELL **98**

4.1. Scripts y Funciones del Shell 98

4.1.1. Funciones 101

4.2. Variables del Shell 107

4.2.1. Parámetros de Posición 108

4.2.2. Más Sobre Sintaxis de Variables 112

4.2.3. Añadir una Variable 113

4.3. Variables Compuestas 114

4.3.1. Asignación de Variables Compuestas 114

4.4. Referencias Indirectas a Variables (namerefs) 116

4.5. Operadores de Cadena 117

4.5.1. Sintaxis de los Operadores de Cadena 117

4.5.2. Patrones y Expresiones Regulares 122

4.5.3. Operadores de Coincidencia de Patrones 131

4.5.4. Operadores de Sustitución de Patrones 134

4.5.5. Operadores de nombre de variable 136

4.5.6. Operadores de longitud 137

4.5.7. La variable *.sh.match* 137

4.6. Sustitución de comandos 138

4.7. Ejemplos avanzados: pushd y popd 143

5. CONTROL DE FLUJO **147**

5.1. if/else 148

5.1.1. Estado de salida y retorno 149

5.1.2. Combinaciones de estado de salida 154

5.1.3. Inversión del sentido de prueba 156

5.1.4. Pruebas de estado 156

5.2. for 167

5.3. case 174

5.3.1. Fusionando «cases» 178

5.4. select 179

5.5. while y until 183

5.5.1. break y continue 186

6. OPCIONES DE LA LÍNEA DE COMANDOS Y VARIABLES TIPO-

GRÁFICAS **188**

6.1. Opciones de línea de comandos . . . 188
6.1.1. shift . . . 189
6.1.2. Opciones con argumentos . . . 191
6.1.3. getopts . . . 192
6.2. Variables Enteras y Aritmética . . . 200
6.2.1. Funciones Aritméticas Incorporadas . . . 203
6.2.2. Condiciones Aritméticas . . . 203
6.2.3. Variables y Asignación Aritmética . . . 204
6.3. for aritmétrico . . . 210
6.4. Arrays (matrices) . . . 212
6.4.1. Matrices indexadas . . . 212
6.4.2. Matrices asociativas . . . 215
6.4.3. Operadores de nombre de matriz . . . 217
6.5. typeset . . . 218
6.5.1. Variables locales en funciones . . . 218
6.5.2. Opciones de formato de cadena . . . 219
6.5.3. Opciones de tipo y atributo . . . 223
6.5.4. Opciones de función . . . 225

7. ENTRADA/SALIDA Y PROCESAMIENTO DESDE LA LÍNEA DE COMANDOS **228**

7.1. Redireccionadores de E/S . . . 228
7.1.1. Here-Documents . . . 230
7.1.2. Here-Strings . . . 232
7.1.3. Descriptores de ficheros . . . 233
7.1.4. Nombres de archivos especiales . . . 236
7.2. Cadenas de E/S . . . 238
7.2.1. print . . . 238
7.2.2. printf . . . 240
7.2.3. Especificadores adicionales de printf en el shell de Korn . . . 244
7.2.4. read . . . 245
7.3. Procesamiento de la línea de comandos . . . 257
7.3.1. Expansión de llaves y sustitución de procesos . . . 257
7.3.2. Orden de sustitución . . . 260

7.3.3. Citando . . . 264
7.3.4. eval . . . 270

8. MANEJO DE PROCESOS 280

8.1. ID de proceso y números de trabajo . . . 281
8.2. Control de trabajo . . . 282
8.2.1. Primer plano y segundo plano . . . 282
8.2.2. Suspendiendo un trabajo . . . 285
8.2.3. Desvinculando un trabajo . . . 286
8.3. Señales . . . 286
8.3.1. Señales con Control-Key . . . 287
8.3.2. kill . . . 288
8.3.3. ps . . . 290
8.3.4. kill: La historia completa . . . 294
8.4. trap . . . 295
8.4.1. Trampasy funciones . . . 297
8.4.2. Variables de ID de proceso y ficheros temporales . . . 299
8.4.3. Ignorando señales . . . 301
8.4.4. Restablecimiento de trampas . . . 302
8.5. Corrutinas . . . 303
8.5.1. wait . . . 305
8.5.2. Ventajas y desventajas de las corrutinas . . . 305
8.5.3. Paralelización . . . 306
8.5.4. Corrutinas con tuberías bidireccionales . . . 309
8.5.5. Tuberías Bidireccionales versus Tuberías Estándar . . . 311
8.6. Subprocesos Shell y Subshell . . . 312
8.6.1. Herencia de subprocesos de shell . . . 312
8.6.2. Subshells . . . 312

9. DEPURACIÓN DE PROGRAMAS DEL SHELL 316

9.1. Ayudas básicas para depuración . . . 317
9.1.1. Establecer opciones . . . 317
9.1.2. Señales falsas . . . 320
9.1.3. Funciones de disciplina . . . 325
9.2. Un depurador para Korn Shell . . . 327

9.2.1. Estructura del depurador . . . 327
9.2.2. El preámbulo . . . 329
9.2.3. Funciones del depurador . . . 331
9.2.4. Condiciones de interrupción . . . 337
9.2.5. Ejemplo de sesión kshdb . . . 341
9.2.6. Ejercicios . . . 343

10. ADMINISTRACIÓN DE KORN SHELL 350

10.1. Instalación del shell Korn como shell estándar . . . 350
10.2. Personalización del entorno . . . 352
10.2.1. umask . . . 352
10.2.2. Tipos de personalización global . . . 355
10.3. Personalización de los modos de edición . . . 357
10.4. Funciones de seguridad del sistema . . . 359
10.4.1. Consejos para Scripts de Shell Seguros . . . 359
10.4.2. Caballos de Troya . . . 364

A. SHELLS RELACIONADOS 368

A.1. El Shell de Bourne . . . 369
A.2. El Shell Korn de 1988 . . . 373
A.3. El Estándar del Shell POSIX IEEE 1003.2 . . . 376
A.4. dtksh . . . 380
A.5. tksh . . . 382
A.6. pdksh . . . 384
A.7. bash . . . 385
A.8. zsh . . . 389
A.8.1. Globbing extendido . . . 389
A.8.2. Completado . . . 390
A.8.3. Editor de línea de comandos . . . 392
A.8.4. Indicadores y temas de indicadores . . . 392
A.8.5. Diferencias entre *zsh* y *ksh* . . . 393
A.9. Sustitutos en Plataformas PC . . . 397
A.9.1. Cygwin . . . 398
A.9.2. DJGPP . . . 398
A.9.3. MKS Toolkit . . . 399

A.9.4. Thompson Automation Software Toolkit 399
A.9.5. AT&T UWIN 400

B. INFORMACIÓN DE REFERENCIA **401**
B.1. Opciones de invocación 401
B.2. Comandos integrados y palabras clave 402
B.3. Alias predefinidos 405
B.4. Variables de Shell incorporadas 405
B.5. Operadores de prueba 409
B.6. Opciones 410
B.7. Opciones tipográficas 412
B.8. Aritmética 413
B.9. Comandos del modo Emacs 415
B.10. Comandos del modo de control vi 417
B.11. Uso de *getopts* 421

C. CONSTRUCCIÓN DE KSH A PARTIR DEL CÓDIGO FUENTE **428**
C.1. Sitios web de Korn Shell 428
C.2. Lo que puedes descargar 429
C.3. Construcción de ksh (Shell Korn) 430

D. ACUERDO DE LICENCIA DE CÓDIGO FUENTE DE AT&T ast **432**

Índice de figuras

1.1. El shell es una capa alrededor del sistema operativo Unix 2
1.2. Árbol de directorios y archivos 10

4.1. Formas de ejecutar un script de shell 100
4.2. Las funciones tienen suspropios parámetros de posición 110

5.1. Archivos producidos por un compilador C 162

6.1. Variables locales en funciones 220

7.1. Sustitución de procesos para flujos de datos de entrada y salida 259
7.2. Pasos en el procesamiento de la línea de comandos 260

8.1. Trabajos en segundo plano en múltiples ventanas 292
8.2. E/S de las corrutinas 309

Índice de tablas

1.1. Ejemplos de comandos cd 12
1.2. Comodines básicos 16
1.3. Usando el comodín * 17
1.4. Uso de los comodines de construcción de conjuntos 17
1.5. Utilidades populares de filtrado de datos Unix 21
1.6. Caracteres especiales 28
1.7. Teclas de control 34

2.1. Comandos básicos del modo emacs 40
2.2. Comandos de palabra en modo Emacs 42
2.3. Comandos de línea en modo Emacs 43
2.4. Comandos del modo Emacs para moverse por el fichero histórico 44
2.5. Comandos varios en modo Emacs 47
2.6. Comandos de edición en modo de entrada vi 51
2.7. Comandos básicos del modo de control vi 52
2.8. Comandos para entrar en el modo de entrada de vi 54
2.9. Algunos comandos de borrado en modo vi 55
2.10. Abreviaturas de los comandos de borrado en modo vi 56
2.11. Comandos para cortar y pegar en modo Vi 56
2.12. Comandos del modo de control Vi para buscar en el fichero histórico . . . 58
2.13. Comandos de búsqueda de caracteres en modo Vi 60
2.14. Comandos varios del modo vi 63

3.1. Opciones básicas del shell 77
3.2. Variables del modo de edición 80
3.3. Variables de correo 81
3.4. Variables de estado 92

4.1. Opciones del comando whence 103

4.2. Operadores de sustitución . . . 118
4.3. Operadores de expresiones regulares . . . 124
4.4. Ejemplos de operadores de expresiones regulares . . . 124
4.5. Clases de caracteres POSIX . . . 127
4.6. Operadores de expresión regular del shell frente a egrep/awk . . . 127
4.7. Nuevos operadores de coincidencia de patrones en *ksh93l* y versiones posteriores . . . 129
4.8. Secuencias de escape de expresiones regulares . . . 130
4.9. Operadores de concordancia de patrones . . . 131
4.11. Operadores de sustitución de patrones . . . 134
4.12. Operadores relacionados con el nombre . . . 136
4.13. Ejemplo de pushd/popd . . . 144

5.1. Valores de estado de salida . . . 154
5.2. Operadores de comparación de cadenas . . . 157
5.3. Operadores de atributos de archivos . . . 163
5.4. Operadores aritméticos de prueba . . . 167

6.1. Opciones Populares del Compilador de C . . . 197
6.2. Operadores Aritméticos . . . 201
6.3. Funciones aritméticas incorporadas . . . 203
6.4. Asignaciones de Expresiones Enteras de Muestra . . . 205
6.5. Operadores relacionados con nombres de arrays . . . 217
6.6. Opciones de formato de cadena de `typeset` . . . 220
6.7. Ejemplos de opciones de formato de cadena de `typeset` . . . 221
6.8. Opciones de tipo y atributo de `typeset` . . . 223
6.9. Opciones de función de `typeset` . . . 225

7.1. Redireccionadores de E/S . . . 229
7.2. Secuencias de escape de print . . . 238
7.3. Opciones de print . . . 239
7.4. Especificadores de formato utilizados en *printf* . . . 241
7.5. Significado de la precisión . . . 242
7.6. Banderas para printf . . . 243
7.7. Opciones de `read` . . . 254
7.8. Ejemplos de reglas de comillas . . . 264

7.9. Secuencias de escape de cadenas 269

8.1. Formas de referirse a trabajos en segundo plano 285
8.2. Opciones para kill 294

9.1. Opciones de depuración 318
9.2. Señales simuladas 321
9.3. Funciones de disciplina predefinidas 326
9.4. Variables especiales para usar en funciones de disciplina 326
9.5. Comandos de *kshdb* 333

10.1. Opciones de recursos `ulimit` 354
10.2. Variables de edición especiales 357

A.1. Equivalentes de patrones *ksh/zsh* 396

C.1. Arquitecturas admitidas para programas AST 430

CAPÍTULO 1

CONCEPTOS BÁSICOS DE KORN SHELL

Usted ha utilizado su ordenador para realizar tareas sencillas, como invocar sus programas de aplicación favoritos, leer su correo electrónico y quizás examinar e imprimir archivos. Sabe que su máquina ejecuta el sistema operativo Unix, o quizá lo conozca bajo algún otro nombre, como Solaris, HP-UX, AIX o SunOS. (O puede que esté utilizando un sistema como GNU/Linux o uno de los sistemas derivados de 4.4-BSD que no está basado en el código fuente original de Unix). Pero aparte de eso, puede que no hayas pensado demasiado en lo que ocurre dentro de la máquina cuando escribes un comando y pulsas ENTER.

Es cierto que varias capas de eventos tienen lugar cada vez que introduces un comando, pero vamos a considerar sólo la capa superior, conocida como shell. En términos generales, un shell es cualquier interfaz de usuario para el sistema operativo Unix, es decir, cualquier programa que toma la entrada del usuario, la traduce en instrucciones que el sistema operativo puede entender, y transmite la salida del sistema operativo de vuelta al usuario.

Hay varios tipos de interfaz de usuario. El shell de Korn pertenece a la categoría más común, conocida como interfaces de usuario basadas en caracteres. Estas interfaces aceptan líneas de comandos textuales que el usuario escribe; normalmente producen una salida basada en texto. Otros tipos de interfaz incluyen las ahora comunes interfaces gráficas de usuario (GUI), que añaden la capacidad de mostrar gráficos arbitrarios (no sólo caracteres de máquina de escribir) y de aceptar entradas de ratones y otros dispositivos señaladores, interfaces de pantalla táctil (como las que se ven en algunos cajeros automáticos), etc.

1.1. ¿Qué es un Shell?

El trabajo del shell, entonces, es traducir las líneas de comando del usuario en instrucciones del sistema operativo. Por ejemplo, considere esta línea de comando:

```
sort -n phonelist > phonelist.sorted
```

Esto significa: «Ordena las líneas del archivo *phonelist* en orden numérico, y pon el resultado en el archivo *phonelist.sorted*» Esto es lo que hace el shell con este comando:

1. Divide la línea en los trozos `sort`, `-n`, `phonelist`, `>`, y `phonelist.sorted`. Estos trozos se llaman *palabras*.
2. Determina el propósito de las palabras: `sort` es un comando; `-n` y `phonelist` son argumentos; `>` y `phonelist.sorted`, en conjunto, son instrucciones de E/S.
3. Establece la E/S según `> phonelist.sorted` (salida al archivo *phonelist.sorted*) y algunas instrucciones estándar implícitas.
4. Busca el comando *sort* en un archivo y lo ejecuta con la opción *-n* (orden numérico) y el argumento *phonelist* (nombre de archivo de entrada).

Por supuesto, cada paso implica realmente varios subpasos, y cada subpaso incluye una instrucción particular al sistema operativo subyacente.

Recuerde que el shell en sí mismo no es Unix - sólo la interfaz de usuario para él. Esto se ilustra en la Figura 1.1. Unix es uno de los primeros sistemas operativos en hacer la interfaz de usuario independiente del sistema operativo.

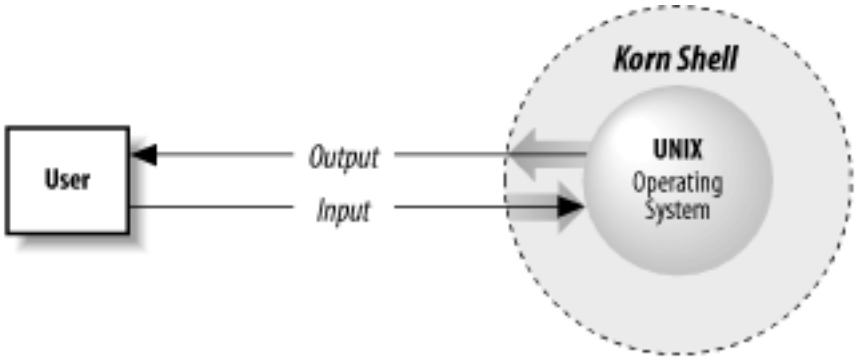

Figura 1.1: El shell es una capa alrededor del sistema operativo Unix

1.2. Alcance de Este Libro

En este libro, aprenderá sobre el shell de Korn, que es el más reciente y potente de los shells distribuidos con los sistemas Unix comerciales. Hay dos formas de utilizar el shell de Korn: como interfaz de usuario y como entorno de programación.

Este capítulo y el siguiente cubren el uso interactivo. Estos dos capítulos deberían darle

suficiente información para usar el shell con confianza y de forma productiva para la mayoría de sus tareas diarias.

Después de haber utilizado el shell durante un tiempo, sin duda encontrará ciertas características de su entorno (la «apariencia» del shell) que le gustaría cambiar y tareas que le gustaría automatizar. El Capítulo 3 muestra varias formas de hacerlo.

El Capítulo 3 también le prepara para la programación del shell, la cual se cubre en su mayor parte en los Capítulos 4 a 6. No necesita tener ninguna experiencia en programación para entender estos capítulos y aprender la programación del shell. El Capítulo 7 y el Capítulo 8 dan descripciones más completas de las capacidades de E/S y de manejo de procesos del shell, y el Capítulo 9 discute varias técnicas para encontrar y eliminar problemas en sus programas del shell.

Aprenderá mucho sobre el shell de Korn en este libro; también aprenderá sobre las utilidades de Unix y el funcionamiento del sistema operativo Unix en general. Es posible convertirse en un virtuoso programador de shell sin ninguna experiencia previa en programación. Al mismo tiempo, hemos evitado cuidadosamente bajar más allá de un cierto nivel de detalle sobre los aspectos internos de Unix. Mantenemos que no deberías ser un experto en el funcionamiento interno para usar y programar el shell de forma efectiva, y no nos detendremos en las pocas características del shell que están pensadas específicamente para los programadores de sistemas de bajo nivel.

1.3. Historia de los Shells de Unix

La independencia del shell del sistema operativo Unix en sí mismo ha llevado al desarrollo de docenas de shells a lo largo de la historia de Unix, aunque sólo unos pocos han logrado un uso generalizado.

El primer intérprete de comandos importante fue el intérprete de comandos Bourne (llamado así por su inventor, Stephen Bourne); se incluyó en la primera versión ampliamente popular de Unix, la versión 7, a partir de 1979. El shell de Bourne se conoce en el sistema como sh. Aunque Unix ha pasado por muchos, muchos cambios, el shell de Bourne sigue siendo popular y esencialmente sin cambios. Varias utilidades y funciones de administración de Unix dependen de él.

El primer shell alternativo ampliamente utilizado fue el shell C, o *csh*. Fue escrito por Bill Joy en la Universidad de California en Berkeley como parte de la versión de Unix de la

Berkeley Software Distribution (BSD) que salió un par de años después de la versión 7. Está incluido en casi todas las versiones recientes de Unix. (Una variante popular es el llamado Twenex *csh*, *tcsh*).

El shell C recibe su nombre del parecido de sus comandos con las declaraciones del lenguaje de programación C, lo que hace que el shell sea más fácil de aprender para los programadores de los sistemas Unix. Soporta una serie de características del sistema operativo (por ejemplo, el control de trabajos; véase el Capítulo 8) que antes eran exclusivas de BSD Unix pero que ahora han migrado a casi todas las demás versiones modernas. También tiene algunas características importantes (por ejemplo, los alias; véase el Capítulo 3) que facilitan su uso en general.

1.3.1. El Shell de Korn

El shell de Korn, o *ksh*, fue inventado por David Korn de AT&T Bell Laboratories a mediados de la década de 1980. Es casi totalmente compatible con el shell de Bourne,[1] lo que significa que los usuarios del shell de Bourne pueden utilizarlo inmediatamente, y todas las utilidades del sistema que utilizan el shell de Bourne pueden utilizar el shell de Korn en su lugar. De hecho, algunos sistemas tienen el shell de Korn instalado como si fuera el shell de Bourne.

El intérprete de comandos Korn comenzó su vida pública en 1986 como parte del «Toolchest Experimental» de AT&T, lo que significa que su código fuente estaba disponible a muy bajo costo para cualquiera que estuviera dispuesto a usarlo sin soporte técnico y con el conocimiento de que todavía podría tener algunos errores. Finalmente, los Laboratorios de Sistemas Unix (USL) de AT&T decidieron darle soporte completo como utilidad Unix. A partir de la versión de Unix de USL llamada System V Release 4 (SVR4 para abreviar, 1989), se distribuyó con todos los sistemas Unix de USL, todas las versiones de Unix de terceros derivadas de SVR4 y muchas otras versiones.

A finales de 1993, David Korn publicó una versión más reciente, conocida popularmente como *ksh93*. Esta versión se distribuye con muchos sistemas Unix comerciales como parte del Entorno Común de Escritorio (CDE), normalmente como el «shell de Korn de escritorio», */usr/dt/bin/dtksh*.

Aunque el propio Unix ha cambiado de propietario varias veces desde entonces, David Korn permaneció en los Laboratorios Bell hasta 1996, cuando AT&T (voluntariamente,

[1] Con algunas excepciones extremadamente pequeñas. Vea el Capítulo 10 para la única importante.

esta vez) se dividió en AT&T Corporation, Lucent Technologies y NCR. En ese momento, se trasladó a AT&T Research Laboratories desde Bell Labs (que siguió formando parte de Lucent). Aunque tanto Lucent como AT&T conservaron todos los derechos del shell de Korn, todas las mejoras y cambios provienen ahora de David Korn en AT&T.

El 1 de marzo de 2000, AT&T liberó el código fuente de *ksh93* bajo una licencia de tipo Open Source. La obtención del código fuente se discute en el Apéndice C, y la licencia se presenta en el Apéndice D.

Este libro se centra en la versión 2001 de *ksh93*. Ocasionalmente, el libro señala una diferencia significativa entre las versiones de 1993 y 1988. Cuando es necesario, las distinguimos como *ksh93* y *ksh88*, respectivamente. En el Apéndice A se describen las diferencias entre las versiones de 1988 y 1993 de forma ordenada, y en ese apéndice también se resumen brevemente otras versiones del shell.

1.3.2. Características del Shell Korn

Aunque el shell de Bourne sigue siendo conocido como el shell «estándar», el shell de Korn también es popular. Además de su compatibilidad con el shell de Bourne, incluye las mejores características del shell C, así como varias ventajas propias. También se ejecuta de forma más eficiente que cualquier shell anterior.

Los modos de edición de la línea de comandos del shell de Korn son las características que suelen atraer a la gente al principio. Con la edición de la línea de comandos, es mucho más fácil volver atrás y corregir errores que con el mecanismo de historial del shell C – y el shell de Bourne no permite hacer esto en absoluto.

La otra característica principal del shell de Korn que está pensada principalmente para los usuarios interactivos es el control de trabajos. Como se explica en el Capítulo 8, el control de trabajos le da la capacidad de parar, iniciar y pausar cualquier número de comandos al mismo tiempo. Esta característica fue tomada casi literalmente del shell C.

El resto de las ventajas importantes del shell de Korn están destinadas principalmente a los personalizadores y programadores del shell. Tiene muchas opciones y variables nuevas para la personalización, y sus características de programación se han ampliado significativamente para incluir la definición de funciones, más estructuras de control, expresiones regulares y aritmética incorporadas, matrices asociativas, variables estructuradas, control avanzado de E/S, y más.

1.4. Cómo Obtener el Shell Korn de 1993

Este libro cubre la versión de 1993 del shell de Korn. Gran parte de lo que se cubre es exclusivo de ese shell; un subconjunto de lo que es exclusivo se aplica sólo a las versiones recientes disponibles directamente de AT&T. Para hacer un mejor uso de este libro, debería estar usando el shell de Korn de 1993. Utilice la siguiente secuencia de instrucciones para determinar qué shell tiene actualmente y si el shell de Korn de 1993 existe en su sistema, y para hacer que el shell de Korn de 1993 sea su shell de inicio de sesión.

1. Determine qué shell está utilizando. La variable SHELL indica su shell de inicio de sesión. Entre en su sistema y escriba *echo $SHELL* en el prompt. Verá una respuesta que contiene sh, *csh*, o *ksh*; estos denotan los shells Bourne, C, y Korn, respectivamente. (También es muy probable que esté utilizando un shell de terceros como bash o tcsh). Si la respuesta es *ksh*, vaya al paso 3. De lo contrario, continúe con el paso 2.

2. Vea si existe alguna versión de *ksh* en su sistema en un directorio estándar. Escriba *ksh*. Si eso funciona (imprime un prompt $), tiene una versión del shell de Korn; continúe con el paso 3. De lo contrario, continúe con el paso 5.

3. Compruebe la versión. Escriba *echo $.sh.version*. Si eso imprime una versión, está todo listo; omita el resto de estas instrucciones. De lo contrario, continúe con el paso 4.

4. No tiene la versión 1993 del shell de Korn. Para saber qué versión tienes, escribe el comando *set -o emacs*, y luego presiona CTRL-V. Esto te dirá si tienes la versión de 1988 o el shell de Korn de dominio público. En cualquier caso, continúe con el paso 5.

5. Escribe el comando `/usr/dt/bin/dtksh`. Si esto le da un prompt *$*, usted tiene el Desktop Korn Shell, que está basado en una versión temprana de *ksh93*. Puede utilizar esta versión; casi todo lo que se dice en este libro funcionará. Vaya al paso 7.

6. Necesita descargar una versión ejecutable de *ksh93* o descargar el código fuente y construir un ejecutable a partir de él. Estas tareas se describen en el Apéndice C. Lo mejor sería solicitar la ayuda de su administrador de sistemas para este paso. Una vez que tenga un *ksh93* que funcione, continúe con el paso 7.

7. Instale *ksh93* como su shell de inicio de sesión. Hay dos situaciones; elija la que le convenga:

Sistema monousuario En un sistema monousuario, en el que usted es el administrador, probablemente tendrá que añadir la ruta completa a *ksh93* al archivo */etc/shells* como primer paso. Luego, debería poder cambiar su shell de inicio de sesión escribiendo *chsh nombre-k*, donde *nombre-k* es la ruta completa al ejecutable *ksh93*. Si esto funciona, se te pedirá tu contraseña. Escriba su contraseña, salga y vuelva a entrar para empezar a usar el shell de Korn.

Si *chsh* no existe o no funciona, consulte la página man de *passwd(1)*. Busque las opciones *-e* o *-s* para actualizar la información de su archivo de contraseñas. Utilice lo que sea apropiado para su sistema para cambiar su shell de inicio de sesión.

Si nada de lo anterior funciona, puede recurrir a editar el archivo */etc/passwd* mientras está conectado como root. Si tienes el comando vipw(8), deberías usarlo para editar tu archivo de contraseñas. Si no, edita el archivo manualmente con tu editor de texto favorito.

Sistema multiusuario grande Esta situación es aún más compleja que el caso de un solo usuario. Lo mejor es dejar que el administrador del sistema se encargue de cambiar el shell por ti. La mayoría de las instalaciones grandes tienen un «servicio de ayuda» (accesible por correo electrónico o por teléfono, o ambos) para introducir dichas solicitudes.

1.5. Uso Interactivo del Shell

Cuando usas el shell de forma interactiva, entras en una sesión de inicio de sesión que comienza cuando te conectas y termina cuando sales o presionas CTRL-D.[2] Durante una sesión de inicio de sesión, escribes líneas de comando en el shell; estas son líneas de texto que terminan en ENTER y que escribes en tu terminal o estación de trabajo.[3] Por defecto, el shell te pide cada comando con un signo de dólar, aunque, como verás en el Capítulo 3, el prompt puede ser cambiado.

[2]Puedes configurar tu shell para que no acepte CTRL-D, es decir, que requiera que escribas `exit` para terminar tu sesión. Recomendamos esto, porque CTRL-D es demasiado fácil de escribir por accidente; vea la sección de opciones en el Capítulo 3.

[3]Aunque asumimos que hay pocas personas que todavía usan terminales seriales reales, los sistemas modernos de ventanas proporcionan acceso al shell a través de un emulador de terminal. Por lo tanto, al menos cuando se trata del uso interactivo del shell, el término «terminal» se aplica igualmente a un entorno de ventanas.

1.5.1. Comandos, Argumentos y Opciones

Las líneas de comandos del shell consisten en una o más palabras, que se separan en una línea de comandos por espacios o TABs. La primera palabra de la línea es el comando. El resto (si lo hay) son argumentos (también llamados parámetros) del comando, que son nombres de cosas sobre las que actuará el comando.

Por ejemplo, la línea de comandos `lpr myfile` está formada por el comando *lpr* (imprimir un archivo) y el único argumento *myfile*. *lpr* trata *myfile* como el nombre de un archivo a imprimir. Los argumentos suelen ser nombres de archivos, pero no necesariamente: en la línea de comandos `mail billr`, el programa de correo trata a *billr* como el nombre del usuario al que se enviará un mensaje.

Una opción es un tipo especial de argumento que da al comando información específica sobre lo que debe hacer. Las opciones suelen consistir en un guión seguido de una letra; decimos «suelen» porque se trata de una convención más que de una regla rígida. El comando `lpr -h miarchivo` contiene la opción *-h*, que le dice a *lpr* que no imprima la «página de bandera» antes de imprimir el archivo.

A veces las opciones tienen sus propios argumentos. Por ejemplo, `lpr -P hp3si -h miarchivo` tiene dos opciones y un argumento. La primera opción es *-P hp3si* , que significa «Enviar la salida a la impresora llamada hp3si». La segunda opción y el argumento son como los anteriores.

1.5.2. Ayuda Incorporada (Built-in)

Casi todos los comandos incorporados en *ksh* tienen ayuda «en línea» tanto mínima como más extensa. Si le das a un comando la opción -?, imprime un breve resumen de uso:

```
$ cd -?
Usage: cd [-LP] [directory]
  Or: cd [ options ] old new
```

(Es posible que desee citar la `?`, ya que, como veremos más adelante, es especial para el shell). También puede dar la opción *–man* para imprimir la ayuda en la forma de la tradicional página man de Unix[4] La salida usa secuencias de escape estándar ANSI para producir un cambio visible en la pantalla, representado aquí usando una fuente en negrita:

```
$ cd --man
NAME
```

[4] A partir de *ksh93i*.

```
  cd - change working directory

SYNOPSIS
  cd [ options ] [directory]
  cd [ options ] old new

DESCRIPTION
  cd changes the current working directory of the current shell
  environment.

  In the first form with one operand, if directory begins with /,
  or if the first component is . or .., the directory will be
  changed to this directory...
```

Del mismo modo, la opción *–html* produce la salida en formato HTML para su posterior renderización con un navegador web.

Por último, la opción *–nroff* le permite producir la ayuda de cada comando en forma de entrada `nroff -man`[5], lo que resulta práctico para dar formato a la ayuda para la salida impresa.

Para cumplir con POSIX, algunos comandos no aceptan estas opciones: *echo, false, jobs, login, newgrp, true* y *:*. Para *test*, tienes que escribir `test - -man - -` para obtener la ayuda en línea.

1.6. Ficheros

Aunque los argumentos de los comandos no siempre son ficheros, los ficheros son los tipos de «cosas» más importantes en cualquier sistema Unix. Un archivo puede contener cualquier tipo de información, y hay diferentes tipos de archivos. Cuatro tipos son, con mucho, los más importantes:

Archivos normales También llamados *archivos de texto*; contienen caracteres legibles. Por ejemplo, este libro se creó a partir de varios archivos regulares que contienen el texto del libro más instrucciones de formato DocBook XML legibles por humanos.

Archivos ejecutables También llamados programas; se invocan como comandos. Algunos no pueden ser leídos por humanos; otros - los scripts del shell que examinaremos en este libro - son sólo archivos de texto especiales. El propio shell es un archivo

[5]Todas las opciones de ayuda envían su salida a error estándar (que se describe más adelante en este capítulo). Esto significa que tiene que usar las facilidades del shell que no cubriremos hasta el Capítulo 7 para capturar su salida. Por ejemplo, `cd --man 2>&1 | more` ejecuta la ayuda en línea a través del programa paginador *more*.

ejecutable (no legible por humanos) llamado *ksh*.

Directorios Como carpetas que contienen otros archivos - posiblemente otros directorios (llamados subdirectorios).

Enlaces simbólicos Una especie de «atajo» de un lugar a otro en la jerarquía de directorios del sistema. Veremos más adelante en este capítulo cómo los enlaces simbólicos pueden afectar al uso interactivo del shell de Korn.

1.6.1. Directorios

Repasemos los conceptos más importantes sobre los directorios. El hecho de que los directorios puedan contener otros directorios da lugar a una estructura jerárquica, más popularmente conocida como árbol, para todos los ficheros de un sistema Unix. La Figura 1.2 muestra parte de un árbol de directorios típico; los óvalos son ficheros normales y los rectángulos son directorios.

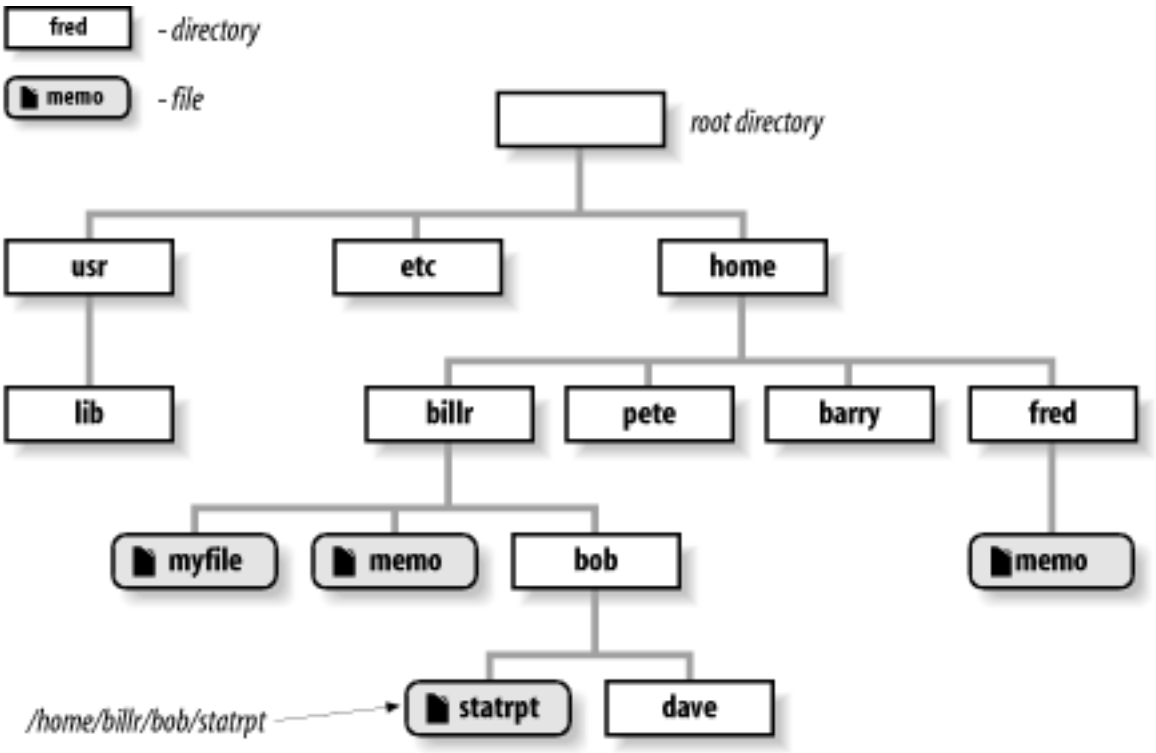

Figura 1.2: Árbol de directorios y archivos

La parte superior del árbol es un directorio llamado «raíz» o «*root*» que no tiene nombre en el sistema[6]. Todos los archivos pueden nombrarse expresando su ubicación en el sistema en relación con la raíz; dichos nombres se construyen enumerando todos los nombres de

[6]La mayoría de los tutoriales introductorios de Unix dicen que *root* tiene el nombre /. Nos quedamos con esta explicación alternativa porque es más consistente lógicamente.

directorio (en orden desde la raíz), separados por barras (/), seguidos del nombre del archivo. Esta forma de nombrar los archivos se denomina *ruta completa (o absoluta).*

Por ejemplo, supongamos que hay un archivo llamado *memo* en el directorio *fred*, que está en el directorio *home*, que está en el directorio raíz. La ruta completa de este fichero es `/home/fred/memo`.

El Directorio de Trabajo

Por supuesto, es molesto tener que utilizar nombres de ruta completos cada vez que necesitas especificar un archivo, por lo que también existe el concepto de *directorio de trabajo* (a veces llamado *directorio actual*), que es el directorio en el que te encuentras en un momento dado. Si se da un nombre de ruta sin barra inicial, la ubicación del archivo se calcula en relación con el directorio de trabajo. Este tipo de nombres de ruta se denominan nombres de ruta relativos; los utilizarás con mucha más frecuencia que los nombres de ruta completos.

Cuando te conectas al sistema, tu directorio de trabajo se establece inicialmente en un directorio especial llamado tu directorio personal (o *de inicio de sesión*). Los administradores de sistemas a menudo configuran el sistema para que el nombre del directorio personal de cada usuario sea el mismo que su nombre de inicio de sesión, y todos los directorios personales están contenidos en un directorio común bajo *root*. En la actualidad, es habitual utilizar */home* como directorio principal para los directorios personales.

Por ejemplo, */home/billr* es un directorio personal típico. Si este es tu directorio de trabajo y das el comando `lp memo`, el sistema busca el archivo memo en */home/billr*. Si tienes un directorio llamado bob en tu directorio personal, y contiene el archivo statrpt, puedes imprimir *statrpt* con el comando `lp bob/statrpt`.

Notación con tilde

Como puedes imaginar, los directorios personales aparecen a menudo en los nombres de ruta. Aunque muchos sistemas están organizados de forma que todos los directorios personales tienen un directorio principal común (como */home*), no debería depender de que esto sea así, ni siquiera debería saber cuál es el nombre absoluto del directorio personal de alguien.

Por lo tanto, el shell de Korn tiene una forma de abreviar los directorios home: basta con preceder el nombre del usuario con una tilde (~). Por ejemplo, puedes referirte al archivo memo en el directorio personal del usuario fred como *~ fred/memo*. Se trata de una ruta

absoluta, por lo que no importa cuál sea tu directorio de trabajo cuando la utilices. Si el directorio personal de fred tiene un subdirectorio llamado bob y el archivo está allí, puedes usar *~fred/bob/memo* como nombre.

Aún más conveniente, una tilde por sí misma se refiere a su propio directorio personal. Puede referirse a un archivo llamado *notas* en su directorio personal como *~/notas* (observe la diferencia entre eso y *~notas*, que el shell trataría de interpretar como el directorio personal del usuario notas). Si *notes* está en tu subdirectorio bob, puedes llamarlo *~/bob/notes*. Esta notación es más útil cuando su directorio de trabajo no está en su árbol de directorios personales, por ejemplo, cuando es algún directorio del «sistema» como */tmp*.

Cambiar los Directorios de Trabajo

Si desea cambiar su directorio de trabajo, utilice el comando *cd*. Si no recuerda su directorio de trabajo, el comando *pwd* indica al shell que lo imprima.

cd toma como argumento el nombre del directorio que desea que se convierta en su directorio de trabajo. Puede ser relativo a su directorio actual, puede contener una tilde, o puede ser absoluto (comenzando con una barra). Si omite el argumento, *cd* cambia a su directorio personal (es decir, es lo mismo que `cd~`).

La Tabla 1.1 muestra algunos ejemplos de comandos *cd*. Cada comando asume que tu directorio de trabajo es */home/billr* justo antes de que se ejecute el comando, y que tu estructura de directorios se parece a la Figura 1.2.

Tabla 1.1: Ejemplos de comandos cd

Comando	**Nuevo directorio de trabajo**
`cd bob`	*/home/billr/bob*
`cd bob/dave`	*/home/billr/bob/dave*
`cd ~/bob/dave`	*/home/billr/bob/dave*
`cd /usr/lib`	*/usr/lib*
`cd ..`	*/home*
`cd ../pete`	*/home/pete*
`cd ~pete`	*/home/pete*
`cd billr pete`	*/home/pete*
`cd illr arry`	*/home/barry*

Los cuatro primeros son sencillos. Los dos siguientes utilizan un directorio especial llamado `..` (dos puntos, pronunciado «dot dot»), que significa «padre de este directorio». Todos los directorios tienen uno de estos; es una forma universal de llegar al directorio situado por encima del actual en la jerarquía, que se denomina directorio padre.

Cada directorio también tiene el directorio especial . (punto simple), que sólo significa «este directorio». Por lo tanto, `cd .` efectivamente no hace nada. Tanto . como .. son en realidad ficheros especiales ocultos en cada directorio que apuntan al propio directorio y a su directorio padre, respectivamente. El directorio raíz es su propio directorio padre.

Los dos últimos ejemplos de la tabla utilizan una nueva forma del comando *cd*, que no está incluida en la mayoría de los shells Bourne. La forma es `cd viejo nuevo`. Toma la ruta completa del directorio de trabajo actual e intenta encontrar la cadena *old* en ella. Si encuentra la cadena, la sustituye por *new* y cambia al directorio resultante.

En el primero de los dos ejemplos, el shell sustituye *pete* por *billr* en el nombre del directorio actual y convierte el resultado en el nuevo directorio actual. El último ejemplo muestra que la sustitución no necesita ser un nombre de archivo completo: al sustituir *arry* por *illr* en */home/billr* se obtiene */home/barry*. (Si no se puede encontrar la cadena antigua en el nombre del directorio actual, el intérprete de comandos imprime un mensaje de error).

Otra característica del comando *cd* del shell de Korn es la forma `cd -`, que cambia al directorio en el que se encontraba antes del actual. Por ejemplo, si empiezas en */usr/lib*, escribes `cd` sin un argumento para ir a tu directorio personal, y luego escribes `cd -`, estarás de vuelta en */usr/lib*.

Enlaces Simbólicos a Directorios

Los sistemas Unix modernos ofrecen *enlaces simbólicos*. Los enlaces simbólicos (a veces llamados *enlaces blandos*) proporcionan una especie de «acceso directo» a los archivos en una parte diferente de la jerarquía de archivos del sistema. Se puede crear un enlace simbólico a un fichero o a un directorio, utilizando nombres de ruta completos o relativos. Cuando se accede a un fichero o directorio mediante un enlace simbólico, Unix «sigue el enlace» hasta el fichero o directorio real.

Los enlaces simbólicos a directorios pueden generar comportamientos sorprendentes. Para explicar por qué, vamos a empezar asumiendo que estás usando el shell de Bourne normal, sh.[7] Ahora, supongamos que nosotros y el usuario fred estamos trabajando juntos en un proyecto, y el directorio primario para el proyecto está bajo su directorio personal, digamos `/home/fred/projects/ important/wonderprog`. Es un nombre de ruta bastante largo, incluso usando la notación con tilde (que no podemos usar en el shell de Bourne, pero eso es otra historia). Para hacer la vida más fácil, vamos a crear un enlace simbólico al

[7] Si tienes un sistema donde el shell de Korn está instalado como /bin/sh, este ejemplo no funcionará

directorio *wonderprog* en nuestro directorio personal:

```
$ sh Utilizar el shell de Bourne
$ cd Asegurarse de que se estamos en nuestro directorio de inicio
$ pwd Mostrar el directorio en el que estamos
/home/billr
Crear el enlace suave
$ ln -s /home/fred/projects/important/wonderprog wonderprog
```

Ahora, cuando escribimos `cd wonderprog`, terminamos en `/home/fred/projects/important/wonderprog`:

```
$ cd wonderprog
$ pwd
/home/fred/projects/important/wonderprog
```

Después de trabajar un rato añadiendo nuevas e importantes características[8] a *wonderprog*, recordamos que necesitamos actualizar el archivo *.profile* en nuestro directorio home. No hay problema: simplemente *cd* de nuevo allí y empezar a trabajar en el archivo, mirándolo primero con *more*.

```
$ cd ..
  Retrocedemos un nivel
$ more .profile
  Observamos .profile
.profile: No such file or directory
```

¿Qué ha pasado? El `cd ..` no nos llevó de vuelta por donde vinimos. En su lugar, subió un nivel en la jerarquía *física* del sistema de archivos:

```
$ pwd
/home/fred/projects/important
```

Este es el «problema» de los enlaces simbólicos: la visión lógica de la jerarquía del sistema de ficheros que presenta un enlace simbólico a un directorio se rompe con la realidad física subyacente cuando se accede al directorio padre.

El shell de Korn funciona de forma diferente. Entiende los enlaces simbólicos y, por defecto, siempre te presenta una vista lógica del sistema de ficheros. No sólo *cd* está integrado en el shell, sino también *pwd*. Ambos comandos aceptan las mismas dos opciones: *-L*, para realizar operaciones lógicas (por defecto), y *-P*, para realizar las operaciones en los directorios reales. Empecemos de nuevo en el shell de Korn:

```
$ cd wonderprog ; pwd
  # cd a traves del enlace simbolico
/home/billr/wonderprog  # La respuesta es la ubicacion logica
```

[8] Las «importantes» son las que desea el departamento de marketing, las necesiten o no los clientes.

```
$ pwd -P
  # Cual es la ubicacion fisica
/home/fred/projects/important/wonderprog # La respuesta es la ubicacion fisica
$ cd .. ; pwd
  # Sube un nivel
/home/billr # El trayecto fue de nuevo logico

$ cd -P wonderprog; pwd
  # Hacer un cd fisico
/home/fred/projects/important/wonderprog # logico es ahora igual a fisico
cd .. ; pwd
  # Vuelve a subir un nivel
/home/fred/projects/important # logico sigue siendo igual a fisico
```

Como se muestra, la opción *-P* de *cd* y *pwd* le permite «evitar» el uso por defecto del posicionamiento lógico del shell de Korn. La mayoría de las veces, sin embargo, el posicionamiento lógico es exactamente lo que quieres.

NOTA: El intérprete de comandos establece las variables PWD y OLDPWD de forma correspondiente cada vez que se realiza un *cd*; los resultados de escribir *pwd* y *print $PWD* deberían ser siempre los mismos.

Como nota no relacionada que redondea la discusión, los sistemas Unix también proporcionan «enlaces duros» (o simplemente *enlaces*) a los archivos. Cada nombre para un fichero se llama enlace; todos los enlaces duros se refieren a los mismos datos en el disco, y si el fichero cambia de nombre, ese cambio se ve al mirarlo desde un nombre diferente. Los enlaces duros tienen ciertas restricciones, que los enlaces simbólicos superan. (Véase *ln(1)* para más información). Sin embargo, no puedes hacer enlaces duros a directorios, así que los enlaces simbólicos son todo lo que importa para *cd* y *pwd*.

1.6.2. Nombres de Archivo y Comodines

A veces es necesario ejecutar un comando en más de un archivo a la vez. El ejemplo más común de este tipo de órdenes es *ls*, que muestra información sobre los ficheros. En su forma más simple, sin opciones ni argumentos, enumera los nombres de todos los archivos del directorio de trabajo excepto los archivos ocultos especiales, cuyos nombres empiezan por un punto (.).

Si le das a *ls* argumentos de nombre de archivo, listará esos archivos, lo cual es un poco tonto: si tu directorio actual tiene los archivos bob y fred, y escribes `ls bob fred`, el sistema simplemente te devolverá los nombres de los archivos.

En realidad, `ls` se usa más a menudo con opciones que le dicen que liste información sobre los ficheros, como la opción *-l* (long), que le dice a *ls* que liste el propietario del fichero, el grupo, el tamaño, la hora de la última modificación y otra información, o *-a* (all), que también lista los ficheros ocultos descritos anteriormente. Pero a veces quieres verificar la existencia de un determinado grupo de archivos sin tener que saber todos sus nombres; por ejemplo, si diseñas páginas web, puede que quieras ver qué archivos de tu directorio actual tienen nombres que terminan en *.html*.

Los nombres de fichero son tan importantes en Unix que el shell proporciona una forma integrada de especificar el patrón de un conjunto de nombres de fichero sin tener que conocer todos los nombres en sí. Puede utilizar caracteres especiales, llamados *comodines*, en los nombres de archivo para convertirlos en patrones. Mostraremos los tres tipos básicos de comodines que soportan los principales shells de Unix, y dejaremos el conjunto de operadores comodín avanzados del shell de Korn para el Capítulo 4. La Tabla 1.2 lista los comodines básicos.

Tabla 1.2: Comodines básicos

Comodín	**Coincidencia**
?	Cualquier carácter
*	Cualquier cadena de caracteres
[set]	Cualquier caracter del conjunto
[!set]	Cualquier carácter no incluido en el conjunto

El comodín `?` coincide con cualquier carácter, de modo que si el directorio contiene los archivos *program.c, program.log* y *program.o*, la expresión `program.?` coincide con *program.c* y *program.o*, pero no con *program.log*.

El asterisco (*) es más potente y se utiliza mucho más; coincide con cualquier cadena de caracteres. La expresión `program.*` coincidirá con los tres archivos del párrafo anterior; los diseñadores web pueden utilizar la expresión `*.html` para coincidir con sus archivos de entrada[9].

La Tabla 1.3 debería darte una mejor idea de cómo funciona el asterisco. Suponga que tiene los archivos *bob, darlene, dave, ed, frank* y *fred* en su directorio de trabajo.

Observe que * puede no significar nada: tanto `*ed` como `*e*` coinciden con *ed*. Observe también que el último ejemplo muestra lo que hace el intérprete de comandos si no puede

[9]Los usuarios de MS-DOS, Windows y OpenVMS deben tener en cuenta que el punto (.) no tiene nada de especial en los nombres de archivo Unix (aparte del punto inicial, que «oculta» el archivo); es simplemente otro carácter. Por ejemplo, ls * lista todos los ficheros del directorio actual; no se necesita *.* como en otros sistemas.

encontrar ninguna coincidencia: simplemente deja la cadena con el comodín intacta.

Tabla 1.3: Usando el comodín *

Expresión	Campos
fr*	frank fred
*ed	ed fred
b*	bob
e	darlene dave ed fred
r	darlene frank fred
*	bob darlene dave ed frank fred
d*e	darlene dave
g*	g*

Los archivos se guardan dentro de los directorios en un orden no especificado; el shell ordena los resultados de cada expansión de comodines. (En algunos sistemas, la ordenación puede estar sujeta a un orden apropiado para la ubicación del sistema, pero que es diferente del orden de cotejo subyacente de la máquina. Los tradicionalistas de Unix pueden utilizar `export LANG=C` para obtener el comportamiento al que están acostumbrados).

El comodín restante es la construcción set. Un conjunto es una lista de caracteres (por ejemplo, abc), un rango inclusivo (por ejemplo, a-z) o una combinación de ambos. Si desea que el guión forme parte de una lista, sólo tiene que ponerlo en primer o último lugar. La Tabla 1.4 (que asume un entorno ASCII) debería explicar las cosas más claramente.

Tabla 1.4: Uso de los comodines de construcción de conjuntos

Expresión	Coincidencia
`[abc]`	a, b, ó c
`[.,;]`	punto, coma o punto y coma
`[-_]`	Guión medio o guión bajo
`[a-c]`	a, b ó c
`[a-z]`	Todas las letras minúsculas
`[!0-9]`	Todos los no dígitos
`[0-9!]`	Todos los dígitos y signos de exclamación
`[a-zA-Z]`	Todas las letras mayúsculas y minúsculas
`[a-zA-Z0-9_-]`	Todas las letras, todos los dígitos, guión bajo y guión

En el ejemplo del comodín original, `program.[co]` y `program.[a-z]` coinciden con *program.c* y *program.o*, pero no con *program.log*.

Un signo de exclamación tras el corchete izquierdo permite «negar» un conjunto. Por ejemplo, `[!.;]` coincide con cualquier carácter excepto punto y coma; `[!a-zA-Z]` coincide con cualquier carácter que no sea una letra.

La notación de rango es útil, pero no debe hacer demasiadas suposiciones sobre los caracteres que se incluyen en un rango. Por lo general, es seguro utilizar un rango para letras

mayúsculas, minúsculas, dígitos o cualquier subrango de éstos (por ejemplo, `[f-q]`, `[2-6]`). No utilices rangos en caracteres de puntuación o letras mixtas: por ejemplo, no se debe confiar en que `[a-Z]` y `[A-z]` incluyan todas las letras y nada más. El problema es que esos rangos no son del todo portables entre distintos tipos de ordenadores[10].

Otro problema es que los sistemas modernos admiten diferentes configuraciones regionales, que son formas de describir cómo funciona el juego de caracteres local. En la mayoría de los países, el conjunto de caracteres por defecto de la configuración regional es diferente al del ASCII plano. En el Capítulo 4, le mostramos cómo usar expresiones POSIX entre corchetes para denotar letras, dígitos, puntuación y otros tipos de caracteres de una forma portable.

El proceso de hacer coincidir expresiones que contienen comodines con nombres de archivo se denomina *expansión de comodines*. Este es sólo uno de los varios pasos que da el shell al leer y procesar una línea de órdenes; otro que ya hemos visto es la *expansión de tildes*, en la que las tildes se sustituyen por directorios de inicio cuando procede. Veremos otros en capítulos posteriores, y los detalles completos del proceso se enumeran en el Capítulo 7.

Sin embargo, es importante tener en cuenta que los comandos que ejecutas sólo ven los resultados de la expansión de comodines. (De hecho, esto es cierto para todas las expansiones.) Es decir, sólo ven una lista de argumentos, y no tienen conocimiento de cómo surgieron esos argumentos. Por ejemplo, si escribe `ls fr*` y sus archivos son como los descritos anteriormente, entonces el shell expande la línea de comandos a `ls fred frank` e invoca el comando *ls* con los argumentos *fred* y *frank*. Si escribe `ls g*`, entonces (porque no hay coincidencia) *ls* recibirá la cadena literal `g*` y se quejará con el mensaje de error, `g* no encontrado`. [11] (Es probable que el mensaje real varíe de un sistema a otro).

He aquí otro ejemplo que le ayudará a entender por qué esto es importante. Supongamos que eres un programador en C. Esto significa que trabajas con archivos cuyos nombres terminan en `.c` (programas, también conocidos como archivos fuente), `.h` (archivos de cabecera para programas) y `.o` (archivos de código objeto que no son legibles por humanos), así como otros archivos.

Supongamos que quieres listar todos los archivos fuente, objeto y de cabecera de tu direc-

[10] En concreto, los rangos dependen del esquema de codificación de caracteres que utilice tu ordenador. La gran mayoría utiliza ASCII, pero los mainframes IBM utilizan EBCDIC. (En realidad, en los sistemas EBCDIC, ni siquiera las mayúsculas y minúsculas forman un rango contiguo).

[11] Esto es diferente del mecanismo de comodines del shell de C, que imprime un mensaje de error y no ejecuta el comando en absoluto.

torio de trabajo. El comando `ls *.[cho]` hace el truco. El shell expande `*.[cho]` a todos los archivos cuyos nombres terminan en un punto seguido de una `c, h`, u `o` y pasa la lista resultante a *ls* como argumentos.

En otras palabras, *ls* verá los nombres de los ficheros como si se hubieran tecleado individualmente - ¡pero fíjese que no hemos asumido ningún conocimiento de los nombres reales de los ficheros! Dejamos que los comodines hagan el trabajo.

A medida que adquiera experiencia con el shell, reflexione sobre cómo sería la vida sin comodines. Bastante miserable, diríamos nosotros.

Una nota final sobre los comodines. Puede asignar a la variable `FIGNORE` un patrón del shell que describa los nombres de archivo que deben ignorarse durante la comparación de patrones. (Las capacidades completas de patrones del shell se describen más adelante, en el Capítulo 4.) Por ejemplo, *emacs* guarda copias de seguridad de archivos añadiendo un ~ al nombre original. A menudo, no necesitas ver estos archivos. Para ignorarlos, puedes añadir lo siguiente a tu archivo *.profile*:

```
export FIGNORE='*~'
```

Al igual que con la expansión de comodines, la prueba contra FIGNORE se aplica a todos los componentes de un nombre de ruta, no sólo al final.

1.7. Entrada y Salida

El campo del software -en realidad, cualquier campo científico- tiende a avanzar de forma más rápida e impresionante en esas pocas ocasiones en las que a alguien (es decir, no a un comité) se le ocurre una idea pequeña en concepto pero enorme en sus implicaciones. El esquema de entrada y salida estándar de Unix tiene que estar en la lista corta de tales ideas, junto con innovaciones clásicas como el lenguaje LISP, el modelo de datos relacional y la programación orientada a objetos.

El esquema de E/S de Unix se basa en dos ideas deslumbrantemente simples. En primer lugar, la E/S de archivos Unix adopta la forma de secuencias arbitrariamente largas de caracteres (bytes). Por el contrario, los sistemas de archivos más antiguos tienen esquemas de E/S más complicados (por ejemplo, «bloque», «registro», «imagen de tarjeta», etc.). En segundo lugar, todo lo que en el sistema produce o acepta datos se trata como un archivo; esto incluye dispositivos de hardware como unidades de disco y terminales. Los sistemas

antiguos trataban cada dispositivo de forma diferente. Ambas ideas han hecho la vida de los programadores de sistemas mucho más agradable.

1.7.1. E/S Estándar

Por convención, cada programa Unix tiene una única forma de aceptar entrada llamada *entrada estándar*, una única forma de producir salida llamada *salida estándar*, y una única forma de producir mensajes de error llamada *salida de error estándar*, normalmente abreviada como *error estándar*. Por supuesto, un programa puede tener también otras fuentes de entrada y salida, como veremos en el Capítulo 7.

La E/S estándar fue el primer esquema de su clase que se diseñó específicamente para usuarios interactivos, en lugar del antiguo estilo de uso por lotes que normalmente implicaba barajas de tarjetas perforadas. Dado que el shell de Unix proporciona la interfaz de usuario, no debería sorprendernos que la E/S estándar se diseñara para encajar perfectamente con el shell.

Todos los shells manejan la E/S estándar básicamente de la misma manera. Cada programa que invocas tiene los tres canales de E/S estándar configurados para tu terminal o ventana de estación de trabajo, de modo que la entrada estándar es tu teclado, y la salida estándar y el error son tu pantalla o ventana. Por ejemplo, la utilidad de correo imprime mensajes para ti en la salida estándar, y cuando la usas para enviar mensajes a otros usuarios, acepta tu entrada en la entrada estándar. Esto significa que ves los mensajes en tu pantalla y escribes los nuevos en tu teclado.

Cuando sea necesario, puede redirigir la entrada y la salida para que procedan o vayan a un archivo en su lugar. Si quieres enviar el contenido de un archivo preexistente a alguien como correo, redirige la entrada estándar de correo para que lea de ese archivo en lugar de tu teclado.

También puedes conectar programas en una *tubería (pipeline)*, en la que la salida estándar de un programa se alimenta directamente de la entrada estándar de otro; por ejemplo, podrías alimentar la salida de mail directamente al programa *lp* para que los mensajes se impriman en lugar de mostrarse en pantalla.

Esto hace posible el uso de utilidades Unix como bloques de construcción para programas más grandes. Muchos programas de utilidades Unix están pensados para ser utilizados de esta manera: cada uno de ellos realiza un tipo específico de operación de filtrado en el texto

de entrada. Aunque este no es un libro de texto sobre utilidades Unix, son esenciales para el uso productivo del shell. Las utilidades de filtrado más populares se listan en la Tabla 1.5

Tabla 1.5: Utilidades populares de filtrado de datos Unix

Utilidad	**Propósito**
cat	Copia la entrada estándar en la salida estándar
grep	Busca *strings* en la entrada estándar
sort	Ordena las líneas de la entrada estándar
sed	Realiza operaciones de edición en la entrada estándar
tr	Traduce caracteres de la entrada estándar a otros caracteres

Es posible que haya utilizado alguna de estas utilidades antes y se haya dado cuenta de que toman nombres de archivos de entrada como argumentos y producen salida en la salida estándar. Sin embargo, puede que no sepa que todas ellas (y la mayoría de las utilidades de Unix) aceptan la entrada estándar si omite el argumento[12].

Por ejemplo, la utilidad más básica es *cat*, que simplemente copia su entrada a su salida. Si escribe `cat` con un argumento de nombre de archivo, imprimirá el contenido de ese archivo en su pantalla. Pero si lo invocas sin argumentos, leerá la entrada estándar y la copiará en la salida estándar. Pruébelo: *cat* esperará a que escriba una línea de texto; cuando teclee ENTER, *cat* repetirá el texto como un loro. Para detener el proceso, pulse CTRL-D al principio de una línea (vea más abajo lo que significa este carácter). Verás ^D cuando teclees CTRL-D. Esto es lo que debería parecer:

```
$ cat
  Here is a line of text.
Here is a line of text.
This is another line of text.
This is another line of text.
^D
$
```

1.7.2. Redirección de E/S

cat es en realidad la abreviatura de «catenate», es decir, unir. Acepta varios argumentos de nombre de archivo y los copia en la salida estándar. Pero supongamos, por el momento, que *cat* y otras utilidades no aceptan argumentos de nombre de archivo y sólo aceptan la entrada estándar. Como dijimos anteriormente, el shell le permite redirigir la entrada estándar para que provenga de un archivo. La notación `command < filename` hace esto;

[12]Si una determinada utilidad Unix no acepta la entrada estándar cuando se omite el argumento nombre de fichero, pruebe a utilizar - como argumento. Esta es una convención común, aunque no universal.

configura las cosas para que *command* tome la entrada estándar de un archivo en lugar de una terminal.

Por ejemplo, si usted tiene un archivo llamado fred que contiene algún texto, entonces `cat < fred` imprimirá el contenido de fred en su terminal. `sort < fred` ordenará las líneas en el archivo fred e imprimirá el resultado en su terminal (recuerde: estamos pretendiendo que las utilidades no toman argumentos de nombre de archivo).

Del mismo modo, `command > filename` hace que la salida estándar del comando sea redirigida al archivo nombrado. El clásico ejemplo «canónico» de esto es `date > now:` el comando *date* imprime la fecha y hora actual en la salida estándar; el comando anterior la guarda en un archivo llamado *now*.

Las redirecciones de entrada y salida pueden combinarse. Por ejemplo, el comando *cp* se usa normalmente para copiar archivos; si por alguna razón no existiera o estuviera roto, podrías usar *cat* de esta manera:

```
cat < file1 > file2
```

Esto sería similar a `cp archivo1 archivo2`.

Como recurso mnemotécnico, piensa en < y > como «embudos de datos». Los datos entran por el extremo grande y salen por el pequeño.

Cuando se utiliza de forma interactiva, el shell de Korn le permite utilizar un comodín de shell después de un operador de redirección de E/S. Si el patrón coincide *exactamente* con un archivo, ese archivo se utiliza para la redirección de E/S. De lo contrario, el patrón no se modifica y el shell intenta abrir un archivo cuyo nombre sea exactamente el que ha escrito. Además, no es válido intentar una redirección con la cadena nula como nombre de fichero (como podría ocurrir cuando se utiliza el valor de una variable, y la variable resulta estar vacía).

Por último, es tentador utilizar el mismo fichero tanto para la entrada como para la salida:

```
sort < myfile > myfile
```

Esto no funciona. El shell trunca *myfile* cuando lo abre para la salida, y no habrá ningún dato allí para que *sort* lo procese cuando se ejecute.

1.7.3. Tuberías (pipes)

También es posible redirigir la salida de un comando a la entrada estándar de otro comando en ejecución en lugar de a un archivo. La construcción que hace esto se llama tubería, anotada como |. Una línea de comandos que incluye dos o más comandos conectados con tuberías se denomina *pipeline*.

Las tuberías se utilizan muy a menudo con el comando *more*, que funciona igual que *cat* excepto que imprime su salida pantalla a pantalla, haciendo una pausa para que el usuario escriba SPACE (siguiente pantalla), ENTER (siguiente línea), u otros comandos. Si estás en un directorio con un gran número de ficheros y quieres ver detalles sobre ellos, `ls -l | more` te dará un listado detallado pantalla a pantalla.

Los pipelines pueden llegar a ser muy complejos (vea, por ejemplo, la función *lsd* en el Capítulo 4 o la versión pipeline del driver del compilador de C en el Capítulo 7); también pueden combinarse con otros redireccionadores de E/S. Para ver un listado ordenado del fichero *fred* pantalla a pantalla, escriba `sort < fred | more`. Para imprimirlo en lugar de verlo en su terminal, escriba `sort < fred | lp`.

He aquí un ejemplo más complicado. El archivo `/etc/passwd` almacena información sobre las cuentas de usuario en un sistema Unix. Cada línea del archivo contiene el nombre de usuario, el número de identificación de usuario, la contraseña encriptada, el directorio de inicio, el shell de inicio de sesión y otra información. El primer campo de cada línea es el nombre de usuario; los campos están separados por dos puntos (:). Una línea de ejemplo podría tener este aspecto:

```
billr:5Ae40BGR/tePk:284:93:Bill Rosenblatt:/home/billr:/bin/ksh
```

Para obtener un listado ordenado de todos los usuarios del sistema, escriba:

```
cut -d: -f1 < /etc/passwd | sort
```

El comando *cut* extrae el primer campo *(-f1)*, donde los campos están separados por dos puntos *(-d:)*, de la entrada. La tubería completa imprime una lista que se parece a esto:

```
al
billr
bob
chris
dave

ed
frank
...
```

Si desea enviar la lista directamente a la impresora (en lugar de a la pantalla), puede ampliar el proceso de la siguiente manera:

```
cut -d: -f1 < /etc/passwd | sort | lp
```

Ahora deberías ver cómo la redirección de E/S y los pipelines apoyan la filosofía de bloques de construcción de Unix. La notación es extremadamente concisa y poderosa. Igualmente importante, el concepto de tubería elimina la necesidad de archivos temporales desordenados para almacenar la salida de comandos antes de que se alimente a otros comandos.

Por ejemplo, para hacer lo mismo que en la línea de comandos anterior en otros sistemas operativos (suponiendo que hubiera disponibles utilidades equivalentes), se necesitarían tres comandos. En el sistema OpenVMS de Compaq, tendrían este aspecto:

```
$ cut [etc]passwd /d=":" /f=1 /out=temp1
$ sort temp1 /out=temp2
$ print temp2
```

Después de practicar lo suficiente, se encontrará tecleando rutinariamente potentes conductos de comandos que hacen en una línea lo que en otros sistemas operativos requeriría varios comandos (y archivos temporales) para lograrlo.

1.8. Trabajos en Segundo Plano

Las tuberías son en realidad un caso especial de una característica más general: hacer más de una cosa a la vez. cualquier otro sistema operativo comercial no tiene esta capacidad, debido a los rígidos límites que tienden a imponer a los usuarios. Unix, por otro lado, fue desarrollado en un laboratorio de investigación y pensado para uso interno, por lo que hace relativamente poco para imponer límites a los recursos disponibles para los usuarios en un

ordenador - como de costumbre, inclinándose hacia la simplicidad despejada en lugar de la sobrecomplejidad.

«Hacer más de una cosa a la vez» significa ejecutar más de un programa al mismo tiempo. Esto se hace cuando se invoca un pipeline; también se puede hacer iniciando sesión en un sistema Unix tantas veces simultáneamente como se desee. (Si lo intentas en un sistema IBM VM/CMS, por ejemplo, obtendrás un odioso mensaje de «ya ha iniciado sesión»).

El shell también le permite ejecutar más de un comando a la vez durante una única sesión de inicio de sesión. Normalmente, cuando escribes un comando y pulsas ENTER, el shell deja que el comando tenga el control de tu terminal hasta que termine; no puedes ejecutar más comandos hasta que termine el primero. Pero si quieres ejecutar un comando que no requiere la entrada del usuario y quieres hacer otras cosas mientras se ejecuta el comando, pon un ampersand (&) después del comando.

Esto se denomina ejecutar el comando en segundo plano, y un comando que se ejecuta de esta forma se denomina *trabajo en segundo plano*; por el contrario, un trabajo que se ejecuta de la forma normal se denomina *trabajo en primer plano*. Cuando inicias un trabajo en segundo plano, recuperas inmediatamente el prompt del shell, lo que te permite introducir otros comandos.

El uso más obvio de los trabajos en segundo plano son los programas que pueden tardar mucho tiempo en ejecutarse, como *sort* o *gunzip* en archivos grandes. Por ejemplo, supongamos que acabas de cargar en tu directorio un enorme archivo comprimido desde una cinta magnética. Hoy en día, la utilidad *gzip* es la utilidad de compresión de archivos de-facto. *gzip* a menudo logra una compresión del 50 % al 90 % de sus archivos de entrada. Los archivos comprimidos tienen nombres del tipo *nombrearchivo.gz*, donde *nombrearchivo* es el nombre del archivo original sin comprimir. Digamos que el archivo es *gcc-3.0.1.tar.gz*, que es un archivo comprimido que contiene más de 36 MB de código fuente.

Escriba `gunzip gcc-3.0.1.tar.gz &`, y el sistema inicia un trabajo en segundo plano que descomprime los datos «en el lugar» y termina con el archivo *gcc-3.0.1.tar*. Justo después de escribir el comando, verá una línea como esta:

```
[1]     4692
```

seguido de su prompt de shell, lo que significa que puede introducir otros comandos. Estos números te dan formas de referirte a tu trabajo en segundo plano; el Capítulo 8 los explica en detalle.

Puede verificar los trabajos en segundo plano con el comando *jobs*. Para cada trabajo en segundo plano, *jobs* imprime una línea similar a la anterior pero con una indicación del estado del trabajo:

```
[1]  +  Running                 gunzip gcc-3.0.1.tar.gz
```

Cuando el trabajo termine, verás un mensaje como este justo antes de tu prompt de shell:

```
[1] +  Done                     gunzip gcc-3.0.1.tar.gz
```

El mensaje cambia si su trabajo en segundo plano finaliza con un error; de nuevo, consulte el Capítulo 8 para más detalles.

1.8.1. E/S en Segundo Plano

Los trabajos que pongas en segundo plano no deberían hacer E/S a tu terminal. Piénsalo un momento y entenderás por qué.

Por definición, un trabajo en segundo plano no tiene control sobre tu terminal. Entre otras cosas, esto significa que sólo el proceso en primer plano (o, en su defecto, el propio shell) está «escuchando» la entrada de tu teclado. Si un trabajo en segundo plano necesita entrada de teclado, a menudo se quedará ahí sin hacer nada hasta que usted haga algo al respecto (como se describe en el Capítulo 8).

Si un trabajo en segundo plano produce salida por pantalla, la salida simplemente aparecerá en su pantalla. Si está ejecutando un trabajo en primer plano que también produce salida, la salida de los dos trabajos se intercalará de forma aleatoria (y a menudo molesta).

Si quieres ejecutar un trabajo en segundo plano que espera una entrada estándar o produce una salida estándar, la solución obvia es redirigirlo para que venga o vaya a un fichero. La única excepción es que algunos programas producen pequeños mensajes de una línea (advertencias, mensajes de «hecho», etc.); puede que no te importe si éstos se intercalan con cualquier otra salida que estés viendo en un momento dado.

Por ejemplo, la utilidad *diff* examina dos archivos, cuyos nombres se dan como argumentos, e imprime un resumen de sus diferencias en la salida estándar. Si los archivos son exactamente iguales, *diff* no dice nada. Normalmente, usted invoca *diff* esperando ver unas pocas líneas que son diferentes.

diff, al igual que *sort* y *gzip*, puede tardar mucho tiempo en ejecutarse si los ficheros de entrada son muy grandes. Suponga que tiene dos archivos grandes llamados *warandpeace.html*

y *warandpeace.html.old*. El comando `diff warandpeace.html.old warandpeace.html` revela que el autor decidió cambiar el nombre «Ivan» por «Aleksandr» en todo el archivo - es decir, cientos de diferencias, lo que resulta en grandes cantidades de salida.

Si escribe `diff warandpeace.html.old warandpeace.html &`, el sistema le arrojará montones y montones de resultados, que serán muy difíciles de detener - incluso con las técnicas explicadas en el Capítulo 7. Sin embargo, si escribe:

```
diff warandpeace.html.antiguo warandpeace.html > wpdiff &
```

las diferencias se guardarán en el archivo *wpdiff* para que pueda examinarlas más tarde.

1.8.2. Trabajos en Segundo Plano y Prioridades

Las tareas en segundo plano pueden ahorrarte mucho tiempo (o pueden ayudarte a hacer dieta eliminando las excusas para ir corriendo a la máquina de caramelos). Pero recuerda que no hay almuerzo gratis; los trabajos en segundo plano consumen recursos que dejan de estar disponibles para ti o para otros usuarios del sistema. El hecho de que ejecutes varios trabajos a la vez no significa que vayan a funcionar más rápido que si se ejecutaran secuencialmente; de hecho, suele ser peor.

A cada tarea del sistema se le asigna una *prioridad*, un número que indica al sistema operativo cuánta prioridad debe darle a la hora de repartir los recursos (cuanto mayor sea el número, menor será la prioridad). Los comandos en primer plano que se introducen desde el shell suelen tener la misma prioridad estándar. Pero los trabajos en segundo plano, por defecto, tienen una prioridad más baja[13]. En el Capítulo 3 descubrirás cómo puedes anular esta asignación de prioridad para que los trabajos en segundo plano se ejecuten con la misma prioridad que los trabajos en primer plano.

Si estás en un gran sistema multiusuario, ejecutar muchos trabajos en segundo plano puede consumir más recursos de los que te corresponden, y deberías considerar si hacer que tu trabajo se ejecute lo más rápido posible es realmente más importante que ser un buen ciudadano. Por otro lado, si tienes una estación de trabajo dedicada con un procesador rápido y mucha memoria y disco, probablemente tengas ciclos de sobra y no debas preocuparte tanto por ello. De todos modos, el patrón de uso típico de estos sistemas obvia en gran medida la necesidad de procesos en segundo plano: basta con iniciar un trabajo y luego abrir otra ventana y seguir trabajando.

[13]Esta característica se tomó prestada del intérprete de comandos C; no está presente en la mayoría de los intérpretes de comandos Bourne.

nice

Hablando de buena ciudadanía, también hay un comando de shell que te permite bajar la prioridad de cualquier trabajo: el acertadamente llamado *nice*. Si escribes lo siguiente, el comando se ejecutará con una prioridad más baja:

```
nice command
```

Puedes controlar cuánto más bajo es dando a *nice* un argumento numérico; consulta la página *man* para más detalles[14].

1.9. Caracteres Especiales y Citas

Los caracteres `<, >, | y &` son cuatro ejemplos de *caracteres especiales* que tienen significados particulares para el shell. Los comodines que vimos anteriormente en este capítulo `(*, ?, y [...])` también son caracteres especiales.

En la Tabla 1.6 se indican los significados de todos los caracteres especiales sólo dentro de las líneas de comandos del shell. Otros caracteres tienen significados especiales en situaciones específicas, como las expresiones regulares y los operadores de manejo de cadenas que veremos en el Capítulo 3 y el Capítulo 4.

Tabla 1.6: Caracteres especiales

Caracter	Significado	Ver capítulo
~	Directorio *Home*	1
`	Sustitución de comandos (arcaico)	4
#	Comentario	4
$	Expresión variable	3
&	Trabajo en segundo plano	1
*	Cadena comodín	1
(	Inicio subshell	8
)	Fin subshell	8
\	Cita siguiente carácter	1
|	Tubería (Pipe)	1
[	Inicio conjunto caracteres comodín	1

[14] Si eres administrador del sistema y has iniciado sesión como `root`, también puedes utilizar *nice* para aumentar la prioridad de una tarea.

Caracter	Significado	Ver capítulo
]	Fin del juego de caracteres comodín	1
{	Inicio bloque de código	7
}	Fin del bloque de código	7
;	Separador de comandos Shell	3
’	Comilla simple	1
”	Doble comilla	1
<	Redirección de entrada	1
>	Redirección de salida	1
/	Separador de directorio	1
?	Comodín de un solo carácter	1
%	Identificador de nombre/número de trabajo	8

1.9.1. Citando

A veces querrá utilizar caracteres especiales literalmente, es decir, sin sus significados especiales. Esto se denomina *entrecomillado*. Si rodea una cadena de caracteres con comillas simples, despojará a todos los caracteres entre comillas de cualquier significado especial que pudieran tener.

La situación más obvia en la que podrías necesitar entrecomillar una cadena es con la orden *print*, que simplemente toma sus argumentos y los imprime en la salida estándar. ¿Para qué sirve esto? Como verá en capítulos posteriores, el intérprete de comandos realiza bastantes procesos en las líneas de comandos, la mayoría de los cuales implican algunos de los caracteres especiales enumerados en la Tabla 1.6. *print* es una forma de hacer que el resultado de ese proceso esté disponible en la salida estándar.

Pero, ¿qué pasaría si quisiéramos imprimir la cadena, `2 * 3 > 5 is a valid inequality` Supongamos que escribimos esto:

```
print 2 * 3 > 5 is a valid inequality
```

Volverías a tener el prompt del shell, ¡como si no hubiera pasado nada! Pero entonces habría un nuevo archivo, con el nombre 5, conteniendo «2», los nombres de todos los archivos en tu directorio actual, y entonces la cadena `3 is a valid inequality`. Asegúrate de entender por qué[15].

Sin embargo, si escribes

```
print '2 * 3 > 5 is a valid inequality.'
```

el resultado es la cadena, tomada literalmente. No es necesario que cites toda la línea, sólo la parte que contiene caracteres especiales (o caracteres que crees que pueden ser especiales, si quieres estar seguro):

```
print '2 * 3 > 5' is a valid inequality.
```

Esto tiene exactamente el mismo resultado.

Observe que la Tabla 1.6 enumera las comillas dobles (") como comillas simples. Una cadena entre comillas dobles está sujeta a algunos de los pasos que sigue el shell para procesar líneas de órdenes, pero no a todos. (En otras palabras, trata sólo algunos caracteres especiales como especiales.) Verá en capítulos posteriores por qué las comillas dobles son a veces preferibles; el Capítulo 7 contiene la explicación más completa de las reglas del shell para las comillas y otros aspectos del procesamiento de la línea de órdenes. Por ahora, sin embargo, deberías ceñirte a las comillas simples.

1.9.2. Barra invertida

Otra forma de cambiar el significado de un carácter es precederlo de una barra invertida (\). Esto se denomina barra invertida-escapar el carácter. En la mayoría de los casos, cuando se escapa una barra invertida, se entrecomilla el carácter. Por ejemplo:

```
print 2 \* 3 \> 5 is a valid inequality.
```

produce los mismos resultados que si rodeara la cadena con comillas simples. Para utilizar una barra invertida literal, basta con rodearla de comillas ('\') o, mejor aún, con escaparla (\ \).

[15]Esto también debería enseñarte algo sobre la flexibilidad de colocar redireccionadores de E/S en cualquier parte de la línea de comandos, incluso en lugares donde no parecen tener sentido.

He aquí un ejemplo más práctico de cómo entrecomillar caracteres especiales. Algunos comandos Unix toman argumentos que a menudo incluyen caracteres comodín, que necesitan ser escapados para que el shell no los procese primero. El comando más común es *find*, que busca archivos en árboles de directorios completos.

Para utilizar *find*, debe proporcionar la raíz del árbol en el que desea buscar y argumentos que describan las características de los archivos que desea encontrar. Por ejemplo, el comando `find . -name string -print` busca en el árbol de directorios cuya raíz es el directorio actual los archivos cuyos nombres coinciden con la cadena e imprime sus nombres. (Otros argumentos permiten buscar por tamaño del archivo, propietario, permisos, fecha del último acceso, etc.).

Puede utilizar comodines en la cadena, pero debe entrecomillarlos para que el propio comando *find* pueda compararlos con los nombres de los archivos de cada directorio en el que busca. El comando `find . -name '*.c'` buscará todos los archivos cuyo nombre termine en `.c` en cualquier lugar del directorio actual, subdirectorios, sub-subdirectorios, etc.

1.9.3. Comillas dobles

También puede utilizar una barra invertida para incluir comillas dobles dentro de una cadena. Por ejemplo

```
print \"2 \* 3 \> 5\" is a valid inequality.
```

produce el siguiente resultado:

```
"2 * 3 > 5" is a valid inequality.
```

Dentro de una cadena entre comillas dobles, sólo es necesario escapar las comillas dobles:

```
$ print "\"2 * 3 > 5\" is a valid inequality."
"2 * 3 > 5" is a valid inequality.
```

Sin embargo, esto no funcionará con comillas simples dentro de expresiones entrecomilladas. Por ejemplo, `print 'Bob\'s hair is brown'` no dará como resultado `Bob's hair is brown`. Puede evitar esta limitación de varias maneras. Primero, intente eliminar las comillas:

```
print Bob\'s hair is brown
```

Si no hay otros caracteres especiales (como en este caso), esto funciona. De lo contrario, puede utilizar el siguiente comando:

```
print 'Bob'\''s hair is brown'
```

Es decir, '\" (comilla simple, barra invertida, comilla simple, comilla simple) actúa como una comilla simple dentro de una cadena entrecomillada. ¿Por qué? La primera ' de '\" termina la cadena entrecomillada que empezamos con 'Bob, la \' inserta una comilla simple literal, y la siguiente ' inicia otra cadena entrecomillada que termina con la palabra «brown». Si entiendes esto, no tendrás problemas para resolver los otros problemas desconcertantes que surgen de la sintaxis a menudo críptica del shell.

Existe un mecanismo algo más legible, específico de *ksh93*, para los casos en los que necesite entrecomillar comillas simples. Este es el mecanismo de entrecomillado extendido del shell: `$'...'`. Esto se conoce en la documentación de *ksh* como entrecomillado ANSI C, ya que las reglas se parecen mucho a las del estándar ANSI/ISO C. Los detalles completos se proporcionan en el Capítulo 7. A continuación se muestra cómo utilizar el entrecomillado ANSI C para el ejemplo anterior:

```
$ print $'Bob\'s hair is brown'
Bob's hair is brown
```

1.9.4. Continuación de líneas

Una cuestión relacionada es cómo continuar el texto de un comando más allá de una sola línea en la ventana de tu terminal o estación de trabajo. La respuesta es conceptualmente sencilla: basta con citar la tecla ENTER. Después de todo, ENTER no es más que otro carácter.

Puede hacerlo de dos maneras: terminando una línea con una barra invertida o no cerrando una comilla (es decir, incluyendo ENTER en una cadena entrecomillada). Si utiliza la barra invertida, no debe haber nada entre ella y el final de la línea - ni siquiera espacios o TABs.

Tanto si utiliza una barra invertida como una comilla simple, le está diciendo al shell que ignore el significado especial del carácter ENTER. Después de pulsar ENTER, el shell entiende que no ha terminado su línea de comandos (es decir, ya que no ha escrito un ENTER «real»), por lo que responde con un prompt secundario, que es > por defecto, y espera a que termine la línea. Puedes continuar una línea tantas veces como desees.

Por ejemplo, si quiere que el shell imprima la primera frase de *The Return of the Native* de Thomas Hardy, puede escribir esto:

```
$ print A Saturday afternoon in November was approaching the \
> time of twilight, and the vast tract of unenclosed wild known \
> as Egdon Heath embrowned itself moment by moment.
```

O puedes hacerlo así:

```
$ print 'A Saturday afternoon in November was approaching the
> time of twilight, and the vast tract of unenclosed wild known
> as Egdon Heath embrowned itself moment by moment.'
```

Hay una diferencia entre los dos métodos. El primero imprime la frase como una línea larga. El segundo conserva las nuevas líneas incrustadas. Pruebe ambos y verá la diferencia.

1.9.5. Teclas de control

Las teclas de control (las que escribes manteniendo pulsada la tecla CONTROL (o CTRL) y pulsando otra tecla) son otro tipo de caracteres especiales. Normalmente no imprimen nada en la pantalla, pero el sistema operativo interpreta algunos de ellos como comandos especiales. Ya conoces uno de ellos: ENTER es en realidad lo mismo que CTRL-M (pruébalo y verás). Probablemente también hayas utilizado la tecla RETROCESO o SUPR para borrar errores tipográficos en tu línea de comandos.

En realidad, muchas teclas de control tienen funciones que realmente no te conciernen - sin embargo, deberías conocerlas para futuras referencias y en caso de que las escribas por accidente.

Quizás lo más difícil de las teclas de control es que pueden diferir de un sistema a otro. La disposición habitual se muestra en la Tabla 1.7, que lista las teclas de control que soportan las principales versiones modernas de Unix. Tenga en cuenta que CTRL-\ y CTRL-| (control-barra invertida y control-tubo) son el mismo carácter anotado de dos formas diferentes; lo mismo ocurre con DEL y CTRL-?

Puede utilizar el comando *stty(1)* para averiguar cuál es su configuración y cambiarla si lo desea; consulte el Capítulo 8 para más detalles. En sistemas Unix modernos (incluyendo GNU/Linux), use `stty -a` para ver su configuración de teclas de control:

```
$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = ; <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
...
```

La notación ^X significa CTRL-X.

Tabla 1.7: Teclas de control

Tecla de control	**Nombre de stty**	**Descripción de la función**
CTRL-C	intr	Detiene el comando actual
CTRL-D	eof	Detiene el comando actual
CTRL-\ ó CTRL-|	quit	Detiene el comando actual, si CTRL-C no funciona
CTRL-S	stop	Detiene el comando actual, si CTRL-C no funciona
CTRL-Q	start	Reinicia la salida a pantalla
BACKSPACE ó CTRL-H	erase	Borra el último carácter. Esta es la configuración más común
DEL ó CTRL-?	erase	Borra último carácter. Este es un ajuste alternativo común. para el carácter de borrado
CTRL-U	kill	Borra toda la línea de comandos
CTRL-Z	susp	Suspende el comando actual (ver Capítulo 8)
CTRL-R	rprnt	Reimprime los caracteres introducidos hasta el momento

La tecla de control que probablemente utilices más a menudo es CTRL-C, a veces llamada *tecla de interrupción*. Esto detiene - o intenta detener - el comando que se está ejecutando en ese momento. Querrás usarla cuando introduzcas un comando y veas que está tardando demasiado, cuando le des los argumentos equivocados por error, cuando cambies de opinión sobre si quieres ejecutarlo, etc.

A veces CTRL-C no funciona; en ese caso, si realmente quieres detener un trabajo, prueba con CTRL-\. Pero no escriba simplemente CTRL-\; ¡pruebe siempre CTRL-C primero! El Capítulo 8 explica por qué en detalle. Por ahora, basta con decir que CTRL-C da al trabajo en ejecución más oportunidad de limpiar antes de salir, para que los ficheros y otros recursos no queden en estados extraños.

Ya hemos visto un ejemplo de CTRL-D. Cuando estás ejecutando un comando que acepta la entrada estándar de tu teclado, CTRL-D (como el primer carácter en la línea) le dice al proceso que tu entrada ha terminado - como si el proceso estuviera leyendo un archivo y llegara al final del archivo. mail es una utilidad en la que esto sucede a menudo. Cuando estás escribiendo un mensaje, terminas tecleando CTRL-D. Esto le dice a mail que su mensaje está completo y listo para ser enviado. La mayoría de las utilidades que aceptan entrada estándar entienden CTRL-D como el carácter de fin de entrada, aunque muchos de estos programas aceptan comandos como *q, quit, exit,* etc. El propio intérprete de comandos entiende CTRL-D como el carácter de fin de entrada: como vimos anteriormente en este capítulo, normalmente puede finalizar una sesión de inicio de sesión escribiendo CTRL-D en el indicador del intérprete de comandos. Sólo le está diciendo al shell que su entrada de comandos ha terminado.

CTRL-S y CTRL-Q se llaman caracteres de control de flujo. Representan una forma anticuada de detener y reiniciar el flujo de salida de un dispositivo a otro (por ejemplo, del ordenador a tu terminal) que era útil cuando la velocidad de dicha salida era baja. Están bastante obsoletos en estos tiempos de redes locales de alta velocidad y líneas de acceso telefónico. De hecho, en estas últimas condiciones, CTRL-S y CTRL-Q son básicamente una molestia. Lo único que realmente necesitas saber sobre ellas es que si la salida de tu pantalla se «atasca», puede que hayas pulsado CTRL-S por accidente. Escribe CTRL-Q para reiniciar la salida; cualquier tecla que hayas pulsado entretanto tendrá efecto.

El último grupo de caracteres de control te ofrece formas rudimentarias de editar tu línea de comandos. RETROCESO o CTRL-H actúan como una tecla de retroceso (de hecho, algunos sistemas usan las teclas DEL o CTRL-? como «borrar» en lugar de RETROCESO); CTRL-U borra toda la línea y te permite empezar de nuevo. De nuevo, la mayoría de estas funciones son obsoletas[16] En lugar de usarlas, vaya al siguiente capítulo y lea sobre los modos de edición del shell de Korn, que están entre sus características más interesantes.

[16] ¿Por qué se siguen utilizando tantas teclas de control obsoletas? No tienen nada que ver con el shell per se; en su lugar, son reconocidas por el controlador tty, una vieja y vetusta parte de las profundidades inferiores del sistema operativo que controla la entrada y salida a/desde tu terminal. De hecho, es el controlador tty el que entiende CTRL-D y señala el fin de la entrada a los programas que leen desde el terminal, no los propios programas.

CAPÍTULO 2

EDICIÓN DESDE LA LÍNEA DE COMANDOS

Siempre es posible cometer errores cuando se escribe en el teclado de un ordenador, pero quizás aún más cuando se utiliza una shell Unix. La sintaxis del shell Unix es potente, pero concisa, llena de caracteres extraños y no especialmente mnemotécnica, lo que permite construir líneas de órdenes tan crípticas como complejas. Los shells Bourne y C agravan esta situación al ofrecerte formas extremadamente limitadas de editar tus líneas de órdenes.

En particular, no hay forma de recuperar una línea de órdenes anterior para poder corregir un error. Por ejemplo, en el Capítulo 7 veremos líneas de comando complejas como:

```
eval cat \$srcname \| ccom \| optimize \| as \> \$objname
```

Si eres un usuario experimentado del shell de Bourne, sin duda conoces la frustración de tener que volver a escribir líneas como ésta. Puedes usar la tecla de retroceso para editar, pero una vez que pulsas ENTER, ¡se ha ido para siempre!

El intérprete de comandos C ofrece una pequeña mejora a través de su mecanismo de historial, que proporciona algunas formas muy incómodas de editar comandos anteriores. Pero hay más de una persona que se ha preguntado: «¿Por qué no puedo editar mis líneas de comandos Unix de la misma forma que puedo editar texto con un editor?»

Esto es exactamente lo que el shell de Korn te permite hacer. Tiene modos de edición que te permiten editar líneas de comandos con comandos de edición similares a los de los dos editores más populares de Unix, *vi* y Emacs.[1] También proporciona un análogo mucho más extendido del mecanismo de historia del shell de C llamado *hist* (por «historia») que, entre otras cosas, te permite usar tu editor favorito directamente para editar tus líneas de comandos.

[1]Por alguna razón desconocida, la documentación sobre el modo *emacs* ha sido eliminada de las páginas de manual de *ksh(1)* en algunos sistemas Unix. Esto no significa, sin embargo, que el modo no exista o no funcione correctamente.

En este capítulo, discutiremos las características comunes a todas las facilidades comando-historial del intérprete de órdenes Korn; después trataremos cada una de esas facilidades en detalle. Si usas vi o Emacs, puede que quieras leer sólo la sección sobre el modo de emulación para el que uses.[2] Si no usas ni *vi* ni Emacs pero estás interesado en aprender uno de los modos de edición de todas formas, sugerimos el modo *emacs*, porque es más una extensión natural de la capacidad mínima de edición que obtienes con el shell desnudo.

Deberíamos mencionar por adelantado que tanto el modo *emacs* como el *vi* introducen la posibilidad de conflictos con las teclas de control establecidas por la interfaz de terminal de Unix. Recuerde las teclas de control mostradas en el Capítulo 1 en la Tabla 1.7 y el ejemplo de salida del comando *stty*. Las teclas de control mostradas allí anulan sus funciones en los modos de edición.

Durante el resto de este capítulo, le advertiremos cuando un comando de edición choque con la configuración por defecto de una tecla de control de la interfaz de terminal. Pero si usted (o su administrador de sistema) elige personalizar su interfaz de terminal, como mostramos en el Capítulo 8, usted está por su cuenta en lo que concierne a los modos de edición.

2.1. Activando la Edición de Línea de Comandos

Hay dos formas de entrar en uno u otro modo de edición. Primero, puede establecer su modo de edición usando la variable de entorno `VISUAL`. El shell de Korn comprueba si esta variable termina con `vi` o emacs.[3] Una forma excelente de establecer `VISUAL` es poner una línea como la siguiente en su fichero *.profile* o de entorno:

```
VISUAL=$(whence emacs)
```

o

```
VISUAL=$(whence vi)
```

Como se verá en los Capítulos 3 y 4, el comando incorporado *whence* toma el nombre de otro comando como argumento y escribe la ruta completa del comando en la salida estándar; la forma *$(command)* devuelve la salida estándar generada por *command* como un valor de cadena. Así, la línea anterior encuentra la ruta completa de tu editor favorito y

[2] Sacarás el máximo provecho de estas secciones si ya estás familiarizado con el/los editor/es en cuestión. Buenas fuentes para información más completa sobre los editores son Learning the vi Editor por Linda Lamb y Arnold Robbins y Learning GNU Emacs por Debra Cameron, Bill Rosenblatt, y Eric Raymond. Ambos publicados por O'Reilly & Associates.

[3] GNU Emacs se instala a veces como *gmacs* o *gnumacs*.

la almacena en la variable de entorno `VISUAL`. La ventaja de este código es que es portable a otros sistemas, que pueden tener los ejecutables de los editores almacenados en directorios diferentes.

La segunda forma de seleccionar un modo de edición es establecer la opción explícitamente con el comando *set -o*:

```
set -o emacs
```

o

```
set -o vi
```

Los usuarios de vi pueden desear añadir:

```
set -o viraw
```

junto con `set -o vi`. Esto permite completar TAB en versiones recientes de *ksh93*. La sobrecarga adicional, particularmente en sistemas monousuario, es nominal y, en cualquier caso, no es peor que la del modo *emacs*. (A partir de *ksh93n*, la opción *viraw* se activa automáticamente cuando se utiliza el modo vi).

Encontrará que los modos de edición *vi* y *emacs* son buenos emulando los comandos básicos de estos editores, pero no las características avanzadas; su propósito principal es permitirle transferir «hábitos de dedo» de su editor favorito al shell. *hist* es una facilidad poderosa; está pensada principalmente para suplantar al historial del shell C y como una «escotilla de escape» para usuarios de editores que no sean *vi* o Emacs. Por lo tanto, la sección sobre *hist* se recomienda sobre todo a los usuarios del intérprete de comandos C y a aquellos que no utilizan ninguno de los dos editores estándar.

Antes de entrar en detalles, vale la pena mencionar otros dos puntos que se aplican a ambos modos de edición:

- *ksh* indica que una línea es más ancha que su pantalla marcando la última columna de la línea visible con un carácter especial: < indica que hay más texto a la izquierda de lo que se ve actualmente, > indica que hay más texto a la derecha de lo que se ve, y * indica que hay texto a ambos lados de lo que se ve actualmente.

  ```
  print this is a very long line that just runs on and >
  ```

- La personalización de los modos de edición de *ksh93* es posible pero requiere el conocimiento de características avanzadas que aún no hemos cubierto. Consulte el

Capítulo 10 para más detalles.[4]

2.2. El Archivo Histórico

Todas las facilidades del historial de comandos del shell de Korn dependen de un fichero que almacena los comandos a medida que los tecleas. Este fichero es normalmente *.sh_history* en tu directorio home, pero puedes llamarlo como quieras configurando la variable de entorno `HISTFILE` (ver Capítulo 3). Cuando ejecutas uno de los modos de edición del shell de Korn, en realidad estás ejecutando un mini-editor en tu fichero histórico.

Si ejecuta más de una sesión de inicio de sesión a la vez (por ejemplo, más de un *xterm* en una estación de trabajo X Windows), puede encontrar ventajoso mantener un fichero de historial separado para cada sesión de inicio de sesión. Ponga la siguiente línea en su *.profile*:

```
HISTFILE=~/.hist.$(tty | sed 's;.*/;;')
```

Esto crea un archivo de historial cuyo nombre termina con el último componente del nombre del dispositivo de su terminal. Por ejemplo, el nombre del dispositivo de terminal de su ventana podría ser */dev/pts/42*. El comando *sed* elimina todo hasta la última barra, dejando sólo el 42. El archivo histórico se convierte entonces en *~/.hist.com*. El archivo histórico se convierte entonces en ~/.hist.42. Puede eliminar el archivo de historial al cerrar la sesión, como se explica en el Capítulo 4. O puede dejar los archivos, como se explica en el Capítulo 5. O puede dejar los archivos, y su historial estará allí la próxima vez que inicie una ventana en el mismo dispositivo terminal. (Preservar el historial entre sesiones es el objetivo del archivo de historial, después de todo).

Una alternativa atractiva es utilizar un único fichero de historial para todas las ventanas. Cada instancia en ejecución del shell de Korn es lo suficientemente inteligente como para compartir su archivo con otras instancias en ejecución; desde una segunda ventana, puede recuperar y editar los comandos ejecutados en la primera ventana.

Otra variable de entorno, `HISTSIZE`, se puede utilizar para determinar el número máximo de comandos accesibles desde el archivo de historial. El valor predeterminado es 128 (es decir, los 128 comandos más recientes), que debería ser más que suficiente.

[4] El shell de dominio público Korn, *bash* y *zsh* tienen modos de edición personalizables, pero de forma diferente a *ksh93*. Véase el Apéndice A.

2.3. Modo de Edición Emacs

Si eres usuario de Emacs, te resultará muy útil pensar en el modo de edición *emacs* como un Emacs simplificado con una única ventana de una línea. Todos los comandos básicos están disponibles para el movimiento del cursor, cortar y pegar, y buscar.

2.3.1. Comando Básicos

El modo Emacs utiliza teclas de control para las funciones de edición más básicas. Si no estás familiarizado con Emacs, puedes pensar en ellas como extensiones del rudimentario carácter «borrar» (normalmente retroceso o DEL) que Unix proporciona a través de su interfaz a los terminales de los usuarios. De hecho, el modo-emacs averigua cuál es tu carácter de borrado y lo usa como tecla de borrar-atrás. En aras de la consistencia, asumiremos que tu carácter de borrado es DEL a partir de ahora; si es CTRL-H o cualquier otro, necesitarás hacer una sustitución mental. Los comandos más básicos de las teclas de control se muestran en la Tabla 2.1.

Tabla 2.1: Comandos básicos del modo emacs

Comando	**Descripción**
CTRL-B	Retroceder un carácter (sin borrar)
CTRL-F	Avanzar un carácter
DEL	Borrar un carácter hacia atrás
CTRL-D	Borrar un carácter hacia delante
CTRL-Y	Recuperar («yank») el último elemento borrado

ADVERTENCIA: ¡Recuerda que teclear CTRL-D cuando tu línea de comandos está vacía puede cerrar tu sesión!

Los hábitos básicos del modo *emacs* son fáciles de aprender, pero requieren que asimiles un par de conceptos peculiares del editor Emacs.

El primero de ellos es el uso de CTRL-B y CTRL-F para avanzar y retroceder el cursor. Estas teclas tienen la ventaja de ser mnemotécnicas obvias, pero mucha gente prefiere usar las teclas de flecha que hay en casi todos los teclados hoy en día.

Desafortunadamente, el modo *emacs* no usa las teclas de dirección,[5] porque los códigos que transmiten al ordenador no están completamente estandarizados; el modo *emacs* fue diseñado para funcionar en la mayor variedad posible de terminales sin la pesada persona-

[5]De hecho, como se describe en el Apéndice B, a partir de *ksh93h*, si su terminal utiliza secuencias de escape estándar ANSI para las teclas de flecha, puede utilizarlas.

lización que necesita el Emacs completo. Casi los únicos requisitos de hardware del modo *emacs* son que el carácter ESPACIO sobrescriba el carácter sobre el que se escribe, y que RETROCESO se mueva a la izquierda sin sobrescribir el carácter actual.

En modo *emacs*, el punto (a veces también llamado punto) es un lugar imaginario justo a la izquierda del carácter sobre el que está el cursor. En las descripciones de los comandos de la Tabla 2-1, algunos dicen «hacia delante» mientras que otros dicen «hacia atrás». Piensa en adelante como «a la derecha del punto» y atrás como «a la izquierda del punto».

Por ejemplo, digamos que tecleas una línea y, en lugar de teclear ENTER, tecleas CTRL-B y lo mantienes pulsado para que se repita. El cursor se moverá hacia la izquierda hasta situarse sobre el primer carácter de la línea, de esta forma:

```
$ fgrep -l Bob < ~pete/wk/names
```

Ahora el cursor está en la f, y el punto está al principio de la línea, justo antes de la f. Si tecleas DEL, no ocurrirá nada porque no hay caracteres a la izquierda del punto. Sin embargo, si pulsas CTRL-D (el comando «borrar carácter hacia delante») borrarás la primera letra:

```
$ grep -l Bob < ~pete/wk/names
```

El punto todavía está al principio de la línea. Si este fuera el comando deseado, podría pulsar ENTER ahora y ejecutarlo; no necesita mover el cursor de vuelta al final de la línea. Sin embargo, si quisiera, podría teclear CTRL-F repetidamente para llegar allí:

```
$ grep -l Bob < ~pete/wk/names
```

En este punto, teclear CTRL-D no haría nada, pero pulsar DEL borraría la s final. Si tecleas DEL y decides que quieres recuperar la s, pulsa CTRL-Y para recuperarla. Si piensas que este ejemplo es una tontería, tienes razón en este caso concreto, pero ten en cuenta que CTRL-Y deshace el último comando de borrado de cualquier tipo, incluidos los comandos borrar-palabra y borrar-línea que veremos en breve [6].

Si haces varios borrados seguidos, CTRL-Y te devuelve todo lo que has borrado. Su memoria se remonta a la última pulsación que no haya sido un borrado; los borrados no tienen por qué ser del mismo tipo. Por ejemplo, si escribes DEL SPACE DEL SPACE CTRL-D CTRL-K, al escribir CTRL-Y recuperas el resultado de las tres últimas operaciones, pero no el primer borrado.

[6] Los usuarios de Emacs deben tener en cuenta que este uso de CTRL-Y es diferente del editor completo, que no guarda los borrados de caracteres.

2.3.2. Comandos de palabras

Los comandos básicos son realmente todo lo que necesitas para moverte por una línea de comandos, pero un conjunto de comandos más avanzados te permite hacerlo con menos pulsaciones. Estos comandos operan sobre *palabras* en lugar de sobre caracteres sueltos; el modo *emacs* define una palabra como una secuencia de uno o más caracteres alfanuméricos o guiones bajos. (Para el resto de esta discusión, será útil pensar en el guión bajo como una letra, aunque realmente no lo sea).

Los comandos de palabra se muestran en la Tabla 2.2. Mientras que los comandos básicos son todos de un solo carácter, los comandos de palabra consisten en dos pulsaciones de tecla, ESC seguida de una letra. Notará que el comando ESC X, donde X es cualquier letra, a menudo hace para una palabra lo que CTRL-X hace para un solo carácter. La multiplicidad de opciones para borrar-palabra-atrás surge del hecho de que tu carácter de borrado puede ser CTRL-H o DEL.

Tabla 2.2: Comandos de palabra en modo Emacs

Opción	**Significado**
ESC b	Retroceder una palabra
ESC f	Mover una palabra hacia adelante
ESC DEL, ESC h, ESC CTRL-H	Borrar una palabra hacia atrás
ESC d	Borrar una palabra hacia adelante

Volviendo a nuestro ejemplo: si tecleamos ESC b, point retrocede una palabra. Como / no es un carácter alfanumérico, emacs-mode se detiene ahí:

```
$ grep -l Bob < ~pete/wk/names
```

El cursor está sobre la `n` en *names*, y el punto está entre la / y la `n`. Ahora digamos que queremos cambiar el argumento de la opción *-l* de este comando de *Bob* a *Dave*. Necesitamos retroceder en la línea de comandos, así que tecleamos ESC b dos veces más. Esto nos lleva aquí:

```
$ grep -l Bob < ~pete/wk/names
```

Si tecleamos ESC b de nuevo, acabaremos al principio de Bob:

```
$ grep -l Bob < ~pete/wk/names
```

¿Por qué? Recuerde que una palabra se define como una secuencia de caracteres alfanuméricos solamente; por lo tanto < no es una palabra, y la siguiente palabra en la dirección hacia atrás es Bob. Ahora estamos en la posición correcta para borrar Bob, así que tecleamos ESC d y obtenemos:

```
$ grep -l < ~pete/wk/names
```

Ahora podemos escribir el argumento deseado:

```
$ grep -l Dave< ~pete/wk/names
```

El comando CTRL-Y «undelete» recuperará una palabra entera, en lugar de un carácter, si una palabra fue lo último que se borró.

2.3.3. Comandos de línea

Todavía hay formas más eficientes de moverse por una línea de órdenes en modo *emacs*. Unos pocos comandos se ocupan de toda la línea; se muestran en la Tabla 2.3.

Tabla 2.3: Comandos de línea en modo Emacs

Comando	Descripción
CTRL-A	Mover al principio de la línea
CTRL-E	Mover al final de la línea
CTRL-K	Suprimir («kill») hacia delante hasta el final de línea
CTRL-C	Poner en mayúscula el carácter después del punto

CTRL-C es a menudo la tecla de «interrupción» que Unix proporciona a través de su interfaz con tu terminal. Si este es el caso, CTRL-C en modo emacs borrará toda la línea, como si se pulsaran CTRL-A y CTRL-K. En sistemas donde la tecla de interrupción está configurada para otra cosa (a menudo DEL), CTRL-C pone en mayúsculas el carácter actual.

Usar CTRL-A, CTRL-E y CTRL-K debería ser sencillo. Recuerda que CTRL-Y siempre deshará lo último borrado, ya sea de un solo comando de borrado o de varios comandos de borrado seguidos. Si usas CTRL-K, podrían ser bastantes caracteres.

2.3.4. Desplazarse por el archivo histórico

Ahora sabemos cómo movernos eficientemente por la línea de comandos y hacer cambios. Pero eso no resuelve el problema original de recuperar comandos anteriores accediendo al fichero de historial. El modo Emacs tiene varios comandos para hacer esto, resumidos en la Tabla 2.4.

Tabla 2.4: Comandos del modo Emacs para moverse por el fichero histórico

Comando	Descripción
CTRL-P	Ir a la línea anteriora
CTRL-N	Ir a la línea siguiente
CTRL-R	Buscar hacia atrás
ESC <	Ir a la primera línea del historial
ESC >	Ir a la última línea del historial

CTRL-P es, con diferencia, la que usarás más a menudo: es la tecla de «he cometido un error; déjame volver atrás y arreglarlo». Puedes utilizarla tantas veces como desees para desplazarte hacia atrás por el historial. Si quieres volver al último comando que introdujiste, puedes mantener pulsado CTRL-N hasta que el shell de Korn emita un pitido, o simplemente teclear ESC >. Por ejemplo, pulsa ENTER para ejecutar el comando anterior, pero aparece un mensaje de error que le indica que la letra de la opción era incorrecta. Quieres cambiarla sin volver a escribir todo. Primero, teclearías CTRL-P para recuperar el comando incorrecto. Lo recuperas con punto al final:

```
$ grep -l Dave < ~pete/wk/names
```

Después de CTRL-A, ESC f, dos CTRL-F y CTRL-D, tienes:

```
ve < ~pete/wk/names
```

Decides probar con *-s* en lugar de *-l*, así que tecleas `s` y pulsas ENTER. Obtienes el mismo mensaje de error, así que te rindes y buscas en el manual. Descubres que el comando que quieres es *fgrep* – no *grep* – después de todo. Suspiras pesadamente y vuelves a buscar el comando *fgrep* que tecleaste hace una hora. Para ello, tecleas CTRL-R; lo que había en la línea desaparece y es sustituido por ^R. Luego escribes *fgrep*, y ves esto:

```
$ ^Rfgrep
```

Pulsa ENTER, y el intérprete de comandos buscará en el historial una línea que contenga «fgrep». Si no encuentra ninguna, emite un pitido. Pero si encuentra una, la muestra, y tu «línea actual» será esa línea (es decir, estarás en algún lugar en medio del fichero histórico, no al final como es habitual):

```
$ fgrep -l Bob < ~pete/wk/names
```

Escribir CTRL-R sin un argumento (es decir, sólo CTRL-R seguido de ENTER) hace que el shell repita su última búsqueda hacia atrás. Si intentas el comando *fgrep* pulsando ENTER de nuevo, ocurren dos cosas. Primero, por supuesto, el comando se ejecuta. Segundo, la línea de comando ejecutada se introduce en el archivo de historial al final, y su «línea actual» también estará al final. Ya no estarás en medio del fichero histórico.

CTRL-P y CTRL-R son claramente los comandos más importantes del modo *emacs* que tienen que ver con el fichero histórico, y puede que utilices CTRL-N ocasionalmente. Los otros son menos útiles, y sospechamos que se incluyeron principalmente por compatibilidad con el editor Emacs completo.

Los usuarios de Emacs también deberían tener en cuenta que las capacidades de búsqueda «de lujo» del editor completo, como la búsqueda incremental y de expresiones regulares, no están disponibles en el modo *emacs* de el Shell de Korn – con una pequeña excepción: si usas CTRL-R y precedes tu cadena de búsqueda con un ^ (carácter de intercalación), sólo coincidirá con comandos que tengan la cadena de búsqueda al principio de la línea.

2.3.5. Nombre de archivo y finalización y expansión de variable

Una de las funciones más potentes (y normalmente infrautilizadas) del modo *emacs* es su función de *completado de nombres* de archivo, inspirada en funciones similares del editor Emacs completo, el intérprete de comandos C y (originalmente) el antiguo sistema operativo DEC TOPS-20.

La premisa detrás del completado de nombres de archivo es que cuando necesites escribir un nombre de archivo, no deberías tener que escribir más de lo necesario para identificar el archivo sin ambigüedades. Se trata de una función excelente; existe una análoga en el modo vi. Te recomendamos que la domines, ya que te ahorrará bastante tecleo.

Hay tres comandos en modo *emacs* relacionados con la autocompletado de nombres de archivo. El más importante es TAB. (A los usuarios de Emacs les resultará familiar; es lo mismo que completar el minibuffer con la tecla TAB). Cuando escribe una palabra de texto seguida de TAB, el shell de Korn intenta completar el nombre de un fichero en el directorio actual. Entonces puede ocurrir una de estas cuatro cosas:

1. Si no hay ningún fichero cuyo nombre empiece por la palabra, el shell emite un pitido y no ocurre nada más.
2. Si hay exactamente una forma de completar el nombre del archivo, y el archivo es un archivo normal, el shell escribe el resto del nombre del archivo y lo sigue con un espacio para que pueda escribir más argumentos de comando.
3. Si hay exactamente una forma de completar el nombre del archivo, y el archivo es un directorio, el shell completa el nombre del archivo y lo sigue con una barra.
4. Si hay más de una forma de completar el nombre de archivo, el shell lo completa con

el prefijo común más largo entre las opciones disponibles.

Por ejemplo, suponga que tiene un directorio con los archivos *program.c* y *problem.c*. Desea compilar el primero de ellos escribiendo `cc programa.c`. Escriba `cc pr` seguido de TAB. Se trata de un prefijo ambiguo, ya que el prefijo «pro» es común a ambos nombres de archivo, por lo que el intérprete de comandos sólo completa `cc pro`. Necesitas escribir más letras para desambiguar, así que escribes g y pulsas TAB de nuevo. Entonces el intérprete de comandos completa a «`cc program.c`», dejando el espacio extra para que usted escriba otros nombres de archivo u opciones.

Un comando relacionado es ESC *, que expande el prefijo a todas las opciones posibles. ESC * actúa como el carácter comodín estándar * del shell, excepto que expande las opciones para que las veas y no ejecuta el comando. En el ejemplo anterior, si escribes ESC * en lugar de TAB, el shell se expandirá a «`cc problem.c program.c`», Si escribe ESC = en lugar de ESC *, verá una lista numerada de expansiones impresa en el error estándar.

A partir de *ksh93m*, el comando ESC = acepta un prefijo numérico. Cuando se proporciona un prefijo, el shell lo trata como el número de uno de los comandos mostrados por un listado ESC = anterior y completa el nombre de archivo. (Más adelante en este capítulo se proporciona un ejemplo donde se describe la versión en modo vi de este comando).

Cuando se usan TAB, ESC * y ESC = en la primera palabra de la línea de comandos, expanden alias, funciones y comandos. Esta característica tan útil se conoce como *finalización de comandos*.

Por compatibilidad con *ksh88* y versiones de *ksh93* anteriores a *ksh93h*, puede escribir ESC ESC para completar el nombre de archivo y el comando.

A partir de *ksh93l*, los modos de edición entienden las reglas de entrecomillado de *ksh*; las expansiones se ignoran dentro de las comillas. Sin embargo, si ha escrito una comilla inicial pero aún no ha cerrado las comillas, los comandos de finalización funcionan. Además, las tres expansiones funcionan también con nombres de variables. (Las variables se tratan en el Capítulo 4.) Cuando *ksh* ve un $ o ”$ y parte de un nombre de variable, puede usar cualquiera de las tres expansiones para ver qué nombres de variables coinciden con lo que ha escrito.

2.3.6. Comandos misceláneos

Varios comandos misceláneos completan el modo de edición de *emacs*; se muestran en la Tabla 2.5.

Tabla 2.5: Comandos varios en modo Emacs

Comando	**Descripción**
CTRL-J	Igual que ENTER.
CTRL-L	Vuelve a mostrar la línea.
CTRL-M	Igual que ENTER.
CTRL-O	Igual que ENTER, luego muestra la línea siguiente en el archivo histórico.
CTRL-T	Transpone los caracteres a ambos lados del punto. Es como GNU Emacs.[7]
CTRL-U	Repite el siguiente comando cuatro veces.
CTRL-V	Imprime la versión del shell de Korn.
CTRL-W	Borra («wipe») todos los caracteres entre punto y «mark». «Marcar» se trata más adelante en esta sección.
CTRL-X CTRL-E	Invoca un editor – normalmente el programa *emacs* – en el comando actual.
CTRL-X CTRL-X	Intercambia punto y marca.
CTRL-[	Igual que ESC (la mayoría de los teclados).
CTRL-] x	Busca x hacia adelante en la línea actual, donde x es cualquier carácter.
CTRL-@	Marca el punto.
ESC c	Cambia la palabra después del punto a mayúsculas.
ESC l	Cambiar la palabra después del punto a todas las letras minúsculas.
ESC p	Guardar todos los caracteres entre el punto y la marca como si se hubieran borrado.
ESC .	Inserta la última palabra de la línea de comando anterior después del punto.
ESC _	Igual que la entrada anterior.
ESC CTRL-] x	Buscar x hacia atrás, donde x es cualquier carácter.
ESC SPACE	Poner marca en el punto.
ESC #	Antepone # (carácter de comentario) a la línea y la envía al archivo de historial; útil para guardar un comando que se ejecutará más tarde sin tener que volver a escribirlo. Si la línea ya empieza con #, elimina el # inicial y cualquier otro carácter de comentario que siga a las nuevas líneas en un comando multilínea.

Varios de estos comandos pueden entrar en conflicto con las teclas de control de la interfaz

[7] Esta es una diferencia con *ksh88*, que transpone dos caracteres a la derecha del punto y mueve el punto uno hacia adelante. CTRL-T se comporta de forma ligeramente diferente si pones `set -o gmacs` (en lugar de `emacs`) en tu *.profile*. En este caso, transpone los dos caracteres a la izquierda del punto, dejando el punto sin mover. Esta es la única diferencia entre los modos *emacs* y *gmacs*; este último se ajusta a la versión de James Gosling del editor Emacs (también conocido como Unipress Emacs, que ya no está disponible).

de terminal de su sistema. CTRL-U es la tecla por defecto para «kill line» en la mayoría de las versiones de Unix. Los sistemas Unix modernos utilizan CTRL-V y CTRL-W como ajustes por defecto para las funciones de interfaz de terminal «citar siguiente carácter» y «borrar palabra», respectivamente. CTRL-V es particularmente confuso, ya que está pensado para anular otras teclas de control de la interfaz de terminal pero no tiene efecto sobre los comandos de *emacs-mode*. Sin embargo, *emacs-mode* funciona interpretando directamente cada carácter que escribes, por lo que la configuración de *stty* se ignora en gran medida.

Vale la pena discutir algunos comandos misceláneos, aunque no estén entre los comandos más útiles de *emacs-mode*.

CTRL-O es útil para repetir una secuencia de comandos que ya has introducido. Simplemente vuelve al primer comando de la secuencia y pulsa CTRL-O en lugar de ENTER. Esto ejecuta el comando y muestra el siguiente comando en el archivo de historial. Pulsa CTRL-O de nuevo para ejecutar este comando y que aparezca el siguiente. Repite esto hasta que veas el último comando de la secuencia; entonces simplemente pulsa ENTER.

CTRL-U, si no realiza la función de borrado de línea de la interfaz de terminal de tu sistema, repite el siguiente comando cuatro veces. Si tecleas CTRL-U dos veces, el factor de repetición se convierte en 16; para 3 CTRL-Us es 64; y así sucesivamente. CTRL-U es posiblemente más útil cuando navegas a través de tu archivo histórico. Si quieres recordar un comando que introdujiste hace un rato, puedes teclear CTRL-U CTRL-P para retroceder cuatro líneas a la vez a través del archivo histórico; puedes pensar en esto como un «rebobinado rápido» a través de tu historial de comandos.

Otro posible uso de CTRL-U es cuando quieres ir de un extremo a otro de una ruta larga. A diferencia del modo vi, el modo *emacs* no tiene un concepto de «palabra» lo suficientemente flexible como para distinguir entre nombres de ruta y componentes de nombres de archivo. Los comandos de movimiento de palabra de *emacs-mode* (ESC b y ESC f) se mueven a través de un nombre de ruta sólo un componente a la vez, porque *emacs-mode* trata la barra como un separador de palabras. Puedes usar CTRL-U para evitar esta limitación. Si tienes una línea como esta:

```
$ ls -l /a/very/long/pathname/filename
```

y necesitas volver atrás y cambiar «very» por «really», puedes teclear CTRL-U ESC b y tu cursor terminará aquí:

```
$ ls -l /a/very/long/pathname/filename
```

A continuación, puedes hacer el cambio. Primero, deshazte de «very» escribiendo CTRL-U CTRL-D:

```
$ ls -l /a/ / long/pathname/filename
```

A continuación, inserta el nuevo texto:

```
$ ls -l /a/really / long/pathname/filename
```

El uso juicioso de CTRL-U puede ahorrarte algunas pulsaciones, pero teniendo en cuenta la pequeña cantidad de información que manipulas cuando editas líneas de comandos, probablemente no sea una función increíblemente vital. A menudo, mantener pulsada una tecla para repetirla es tan eficaz como CTRL-U. Dado que probablemente tendrás que utilizar el comando *stty* para redefinir la tecla de borrado de línea del controlador de terminal antes de poder utilizar CTRL-U, probablemente sea mejor prescindir de ella.

La marca mencionada en la explicación de CTRL-W debería ser familiar para los usuarios de Emacs, pero su función en *emacs-mode* es un subconjunto de la que tiene en el editor completo. El modo Emacs mantiene un registro del lugar en el que se realizó la última operación de borrado (ya sea un carácter, palabra, línea o lo que sea); este lugar se llama *marca*. Si no se ha borrado nada en la línea actual, la marca por defecto es el principio de la línea. También puede establecer la marca en el lugar donde se encuentra el cursor escribiendo ESC ESPACIO (o, alternativamente, CTRL-@). CTRL-X CTRL-X (CTRL-X pulsado dos veces) hace que el shell de Korn intercambie el punto y la marca, es decir, que mueva el cursor a donde está la marca y vuelva a poner la marca donde estaba el cursor antes de teclear CTRL-X CTRL-X.

El concepto de marca no es extremadamente útil debido a la poca «distancia» que hay que recorrer en las líneas de comandos. Pero si alguna vez tienes que hacer una serie de cambios en el mismo lugar de una línea, CTRL-X CTRL-X te llevará de vuelta allí. En el ejemplo anterior, si quisieras cambiar «really» por «monumentalily», una forma sería escribir CTRL-X CTRL-X para volver al principio de «really»:

```
$ ls -l /a/r eally/long/pathname/filename
```

Luego podrías teclear ESC d para borrar «really» y hacer el cambio. Por supuesto, podrías hacer esto más rápido tecleando ESC DEL en lugar de CTRL-X CTRL-X y ESC d.

De los comandos de cambio de mayúsculas y minúsculas, ESC l (letra ell) es útil cuando pulsas la tecla BLOQ MAYÚS por accidente y no te das cuenta inmediatamente. Como

las palabras en mayúsculas no se usan muy a menudo en el mundo Unix, es posible que no uses ESC c muy a menudo.

Si le parece que hay demasiados sinónimos de ENTER, tenga en cuenta que CTRL-M es en realidad el mismo carácter (ASCII) que ENTER, y que CTRL-J es en realidad lo mismo que nueva línea, que Unix suele aceptar en lugar de ENTER de todos modos.

ESC . y ESC _ son útiles si desea ejecutar varios comandos en un archivo dado. La convención habitual de Unix es que un nombre de fichero es el último argumento de un comando. Por lo tanto, puede ahorrar tecleando simplemente cada comando seguido de SPACE y luego tecleando ESC . o ESC _. Por ejemplo, digamos que quieres examinar un archivo usando *more*, así que tecleas:

```
$ more myfilewithaverylongname
```

Luego decides que quieres imprimirlo, usando el comando de impresión *lp*. Puede evitar teclear el nombre muy largo escribiendo *lp* seguido de un espacio y luego ESC . o ESC _; el shell de Korn inserta *myfilewithaverylongname* por usted.

Si eres un verdadero experto en Emacs y el modo incorporado no te funciona, usa CTRL-X CTRL-E para invocar el programa editor *emacs* en tu línea de comandos. Cuando salgas del editor, si realmente hiciste cambios en el archivo, el shell ejecutará la línea de comandos final.

2.3.7. Expansión de Macros con Alias

A medida que te acostumbres a usar el modo *emacs*, puede que te des cuenta de que hay secuencias de comandos que ejecutas una y otra vez. Escribir estos comandos repetidamente es difícil y una pérdida de tiempo. Es mejor definir una *macro* para ellos. Una macro es un nombre corto que, cuando se introduce, se expande en la secuencia completa de comandos.

El shell de Korn proporciona una función de macro, utilizando el mecanismo de alias (descrito en el siguiente capítulo), que le permite establecer una secuencia de comandos y luego invocarla con un único comando en modo *emacs*. Funciona de la siguiente manera: si defines un alias llamado _*x*, donde x es una letra, entonces cuando tecleas ESC x, *emacs-mode* expande el alias, y lo lee como entrada. El valor del alias puede contener texto normal, comandos *emacs-mode*, o ambos.

Por ejemplo, suponga que quiere que un comando ponga en mayúscula la primera letra de la palabra actual. Podrías definir un alias como sigue

```
alias _C='^[b^C'                    # El valor es ESC b CTRL-C
```

Ahora, cada vez que escriba ESC C, el intérprete de comandos se moverá al principio de la palabra actual (ESC b) y, a continuación, pondrá en mayúscula la letra actual (CTRL-C).

```
$ print here is a word          # Type ESC C
$ print here is a Word
```

2.4. Modo de Edición Vi

Al igual que *emacs-mode*, *vi-mode* crea esencialmente una ventana de edición de una línea en el archivo de historial. El modo vi es popular porque vi es el editor más estándar de Unix. Pero la función para la que fue diseñado vi, escribir programas en C, tiene unos requisitos de edición diferentes a los de los intérpretes de comandos. Como resultado, aunque es posible hacer cosas complejas en vi con relativamente pocas pulsaciones de tecla, las cosas relativamente simples que necesitas hacer en el shell de Korn a veces requieren demasiadas pulsaciones de tecla.

Al igual que *vi*, *vi-mode* tiene dos modos propios: modo de entrada y modo de control. El primero es para teclear comandos (como en el uso normal del shell de Korn); el segundo es para moverse por la línea de comandos y el fichero histórico. Cuando estás en modo de entrada, puedes escribir comandos y pulsar ENTER para ejecutarlos. Además, tiene capacidades mínimas de edición a través de caracteres de control, que se resumen en la Tabla 2.6.

Tabla 2.6: Comandos de edición en modo de entrada vi

Comando	Descripción
DEL	Borrar el carácter anterior
CTRL-W	Borrar palabra anterior (es decir, borrar hasta el espacio en blanco)
CTRL-V	«Citar» el carácter siguiente
ESC	Entrar en modo de control (véase más abajo)

NOTA: Al menos algunos de estos comandos de edición – dependiendo de la versión de Unix que tenga – son los mismos que los proporcionados por los sistemas Unix modernos en la interfaz de terminal. *Vi-mode* usa su caracter «erase» como la tecla «delete previous character» (borrar caracter previo); usualmente se establece en DEL o CTRL-H (BACKSPACE). CTRL-V hace que el siguiente carácter que escriba aparezca en la línea de comandos tal cual; es decir, si es un comando de edición (o un carácter especial como CTRL-D), se le quita su significado especial.

En circunstancias normales, sólo tiene que permanecer en el modo de entrada. Pero si quieres volver atrás y hacer cambios en tu línea de comandos, o si quieres recuperar comandos anteriores, necesitas ir al modo de control. Para ello, pulsa ESC.

2.4.1. Comandos de Modo de Control Simple

En el modo de control tienes a tu disposición una amplia gama de comandos de edición de *vi*. Los más sencillos te mueven por la línea de comandos[8] y se resumen en la Tabla 2.7. El modo vi contiene dos conceptos de «palabra». El más simple es cualquier secuencia de caracteres que no sean espacios en blanco; lo llamaremos *palabra no en blanco*. El otro es cualquier secuencia de sólo caracteres alfanuméricos (letras y dígitos) o cualquier secuencia de sólo caracteres no alfanuméricos; llamaremos a esto una *palabra*.[9]

Tabla 2.7: Comandos básicos del modo de control vi

Comando	Descripción
h	Mover un carácter a la izquierda.
l	Mover un carácter a la derecha.
space	Mover a la derecha un caracter.
w	Mover a la derecha una palabra.
b	Mover una palabra a la izquierda.
W	Mover al principio de la siguiente palabra no en blanco.
B	Mover al principio de la palabra anterior.
e	Mover al final de la palabra actual.
E	Mover al final de la palabra actual no en blanco.
0	Va al principio de la línea.
^	Va al primer carácter no en blanco de la línea.
$	Va al final de la línea.

Todos estos comandos, excepto los tres últimos, pueden ir precedidos de un número que actúa como *contador de repeticiones*. Los dos últimos resultarán familiares a los usuarios de utilidades Unix (como *grep*) que utilizan expresiones regulares, así como a los usuarios de vi.

Veamos algunos ejemplos. Digamos que escribes esta línea y, antes de pulsar ENTER, decides que quieres cambiarla:

```
$ fgrep -l Bob < ~pete/wk/names
```

Como se muestra, su cursor está más allá del último carácter de la línea. Primero, teclea ESC para entrar en modo de control; tu cursor retrocede un espacio de modo que se

[8] Al igual que con el modo *emacs*, desde *ksh93h*, puede utilizar secuencias de teclas de flecha estándar ANSI para moverse hacia adelante y hacia atrás en la línea de comandos, y hacia arriba y hacia abajo dentro de la lista del historial.

[9] Ninguna de estas definiciones es la misma que la definición de palabra en modo *emacs*.

encuentra en la `s`. Luego, si tecleas h, tu cursor retrocede a la `e`. Si tecleas `3h` desde la `e`, terminas en la *n*.

Ahora veremos la diferencia entre los dos conceptos de «palabra». Vuelve al final de la línea tecleando $. Si tecleas b, la palabra en cuestión es «names», y el cursor acaba en la `n`:

```
$ fgrep -l Bob < ~pete/wk/names
```

Si vuelves a teclear `b`, la siguiente palabra es la barra (es una «secuencia» de caracteres no alfanuméricos), por lo que el cursor acaba sobre ella:

```
$ fgrep -l Bob < ~pete/wk/names
```

Sin embargo, si tecleas `B` en lugar de `b`, la palabra no en blanco será el nombre completo de la ruta, y el cursor terminará al principio de la misma, es decir, sobre la tilde:

```
$ fgrep -l Bob < ~pete/wk/names
```

Habría tenido que escribir `b` cuatro veces – o simplemente `4b` – para obtener el mismo efecto, ya que hay cuatro «palabras» en la parte de la ruta a la izquierda de */names: wk, slash, pete* y la tilde inicial.

En este punto, `w` y `W` hacen lo contrario: teclear `w` te lleva sobre la `p`, ya que la tilde es una «palabra», mientras que teclear `W` te lleva al final de la línea. Pero mientras que `w` y `W` te llevan al principio de la siguiente palabra, `e` y `E` te llevan al final de la palabra actual. Así, si tecleas `w` con el cursor sobre la tilde, llegas a:

```
$ fgrep -l Bob < ~pete/wk/names
```

Entonces tecleando *e* te lleva a:

```
$ fgrep -l Bob < ~pete/wk/names
```

Y escribiendo una *w* adicional te lleva a:

```
$ fgrep -l Bob < ~pete/wk/names
```

Por otro lado, `E` te lleva al final de la palabra actual no en blanco – en este caso, el final de la línea. (Aunque a primera vista los comandos pueden parecer no nemotécnicos, generalmente hay cierto orden en la elección de las letras de los comandos. Cada letra de comando suele ser la primera letra de la palabra inglesa correspondiente a la operación. Las minúsculas sirven para las palabras, mientras que las mayúsculas sirven para las palabras que no están en blanco. Entender esto es sin duda más difícil si el inglés no es tu lengua materna, pero eso también se aplica a los comandos del modo *emacs*).

2.4.2. Introducción y cambio de texto

Ahora que ya sabes cómo entrar en el modo de control y moverte por la línea de comandos, necesitas saber cómo volver al modo de entrada para poder hacer cambios y escribir comandos adicionales. Una serie de comandos le llevan del modo de control al modo de entrada; se enumeran en la Tabla 2.8. Todos ellos entran en el modo de entrada de forma un poco diferente. Todos ellos entran en el modo de entrada de forma un poco diferente.

Tabla 2.8: Comandos para entrar en el modo de entrada de vi

Comando	Descripción
i	Texto insertado antes del carácter actual (insertar)
a	Texto insertado después del carácter actual (append)
I	Texto insertado al principio de la línea
A	Texto insertado al final de la línea
r	Sustituir un carácter (no entra en modo de entrada)
R	El texto sobrescribe el texto existente (reemplazar)

Lo más probable es que utilices siempre `i` o `a`, y puede que ocasionalmente utilices `R`. `I` y `A` son abreviaturas de `0i` y `$a` respectivamente. Para ilustrar la diferencia entre `i`, `a` y `R`, digamos que empezamos con nuestra línea de ejemplo:

```
$ fgrep -l Bob < ~pete/wk/names
```

Si escribes `i` seguido de end, obtienes:

```
$ fgrep -l Bob < ~pete/wkend/names
```

Es decir, el cursor siempre aparece sobre `/` antes de *names*. Pero si escribes `a` en lugar de `i`, verás que el cursor se mueve un espacio a la derecha. Entonces, si escribes `nick`, obtendrás

```
$ fgrep -l Bob < ~pete/wk/nicknames
```

Es decir, el cursor está siempre justo después del último carácter que has tecleado, hasta que tecleas ESC para finalizar tu entrada. Por último, si vuelve a la `n` en *names*, escriba `R` en su lugar, y luego escriba `task`, verá:

```
$ fgrep -l Bob < ~pete/wk/tasks
```

En otras palabras, usted estará *reemplazando* (de ahí `R`) en lugar de insertar texto.

¿Por qué `R` mayúscula en lugar de `r` minúscula? Este último es un comando ligeramente diferente, que sustituye sólo un carácter y no entra en modo de entrada. Con `r`, el siguiente carácter sobrescribe el carácter situado bajo el cursor. Así que si empezamos con la línea de comandos original y tecleamos `r` seguido de un punto y coma, obtendremos:

```
$ fgrep -l Bob < ~pete/wk;names
```

Si precedes `r` con un número *N*, te permite reemplazar los siguientes *N* caracteres existentes en la línea – pero aún no entras en modo de entrada. El modo Vi sustituye los `N` caracteres de la línea por `N` copias del carácter que escriba después de la `r`. La `r` minúscula es eficaz para corregir letras de opción erróneas, caracteres de redirección de E/S, puntuación, etc.

2.4.3. Comandos de eliminación

Ahora que sabes cómo introducir comandos y moverte por la línea, necesitas saber cómo borrar. El comando básico de borrado en modo vi es `d` seguido de otra letra. Esta letra determina cuál es la unidad y la dirección de borrado, y corresponde a un comando de movimiento, como los listados previamente en la Tabla 2.7. La Tabla 2.9 muestra algunos ejemplos de uso común.

Tabla 2.9: Algunos comandos de borrado en modo vi

Commando	**Descripción**
dh	Borrar un carácter hacia atrás.
dl	Borrar un carácter hacia delante.
db	Borrar una palabra hacia atrás.
dw	Borrar una palabra hacia delante.
dB	Borrar una palabra no en blanco hacia atrás.
dW	Borrar una palabra no en blanco hacia delante.
d$	Borrar hasta el final de línea.
d0	Borrar hasta el principio de línea.

Estos comandos tienen algunas variaciones y abreviaturas. Si utiliza una `c` en lugar de `d`, entrará en el modo de entrada después de que se realice el borrado. Puede proporcionar un número de repetición antes o después de la `d` (o `c`). La Tabla 2.10 lista las abreviaturas disponibles.

La mayoría de la gente tiende a usar `D` para borrar hasta el final de la línea, `dd` para borrar una línea entera, y `x` (como «retroceso») para borrar caracteres individuales. Si no eres un usuario empedernido de vi, puede que te resulte difícil dominar algunos de los comandos de borrado más esotéricos bajo tus dedos.

Tabla 2.10: Abreviaturas de los comandos de borrado en modo vi

Comando	**Descripción**
D	Equivale a d$ (borrar hasta el final de la línea)
dd	Equivale a 0d$ (borrar toda la línea)
C	Equivale a c$ (borrar hasta el final de la línea, entrar en modo de entrada)
cc	Equivale a 0c$ (borrar toda la línea, entrar en modo de entrada)
s	Equivale a xi (borrar el carácter actual, entrar en modo de entrada)
S	Equivale a cc (borrar toda la línea, entrar en modo de entrada)
x	Equivale a dl (borrar carácter hacia adelante)
X	Equivale a dh (borrar carácter hacia atrás)

Todo buen editor proporciona comandos «undelete» así como comandos de borrado, y *vi-mode* no es una excepción. *Vi-mode* mantiene un *buffer de borrado* que almacena todas las modificaciones al texto en la línea actual solamente (note que esto es diferente del editor vi completo). El comando `u` deshace sólo el último comando de modificación de texto, mientras que `U` deshace todos los comandos de este tipo en la línea actual. Así que si haces un cambio y quieres deshacerlo, teclea `u`; si haces muchos cambios y encuentras que el original se acerca más a lo que quieres, puedes deshacerlo todo tecleando `U`. Un comando relacionado es . (punto), que rehace el último comando de modificación de texto.

También hay una forma de guardar texto en el búfer de borrado sin haberlo borrado en primer lugar: simplemente teclea un comando de borrado pero utiliza `y` («yank») en lugar de d. Esto no modifica nada, pero te permite recuperar el texto yankado tantas veces como quieras más adelante. El comando para recuperar el texto arrancado es `p`, que inserta («pone») el texto en la línea actual a la derecha del cursor. La versión en mayúsculas, P, coloca el texto a la izquierda del cursor. Los distintos comandos de cortar y pegar se resumen en la Tabla 2.11.

Tabla 2.11: Comandos para cortar y pegar en modo Vi

Texto	**Descripción**
y	Corta (guarda) texto, no cambia realmente la línea.
p	Pone el último texto arrancado o borrado en la línea después del cursor.
P	Coloca el último texto arrancado o borrado en la línea anterior al cursor.
u	Deshace el cambio más reciente.
U	Deshacer todos los cambios de la línea.
. (punto)	Rehacer el último cambio en la posición actual del cursor.

Los comandos `d` y `p` son bastante útiles juntos para reorganizar el orden de las opciones o argumentos en una línea de comandos. Por ejemplo, la mayoría de los compiladores C de Unix aceptan una opción *-l* que indica el nombre de una biblioteca a utilizar al enlazar un programa compilado. La opción *-L* especifica un directorio en el que el compilador debe

buscar las bibliotecas, además de buscar en los lugares estándar para las bibliotecas del sistema.

```
cc -o myprog myprog.c -Lmylibdir -lmylib
```

Este comando busca el archivo de biblioteca *libmylib.a* en el directorio *mylibdir* al compilar y enlazar *myprog.c*. Hasta aquí todo bien. El problema es que normalmente la opción *-L* debe aparecer en la línea de comandos antes de la opción *-l*. Supongamos que accidentalmente las has escrito al revés y, por tanto, la compilación ha fallado. Puedes utilizar los comandos `d` y `p` para reordenar las cosas. Empieza por recuperar la línea

```
$ cc -o myprog myprog.c -lmylib -Lmylibdir
```

A continuación, desplázate hasta la última opción con `5w`. A continuación, retroceda hasta el espacio anterior con `h`. Su línea de comandos tiene ahora este aspecto:

```
$ cc -o myprog myprog.c -lmylib-Lmylibdir
```

Escribe `D` para borrar el resto de la línea:

```
$ cc -o myprog myprog.c -lmylib
```

Ahora retrocede hasta el carácter `c` anterior con `Bhh`:

```
$ cc -o myprog myprog.c -lmylib
```

Por último, utilice `p` para insertar la opción desplazada:

```
$ cc -o myprog myprog.c -Lmylibdir -lmylib
```

Luego pulsa ENTER y ya está. Esto parece mucho tecleo. Pero, como veremos pronto, hay comandos adicionales que te permiten buscar caracteres en la línea de comandos, haciendo mucho más fácil moverse por ella. Y si eres un usuario experimentado de *vi*, estarás como en casa.

2.4.4. Desplazarse por el archivo histórico

El siguiente grupo de comandos del modo de control de vi que cubrimos te permite moverte y buscar en tu archivo de historial. Esta es la funcionalidad más importante que te permite volver atrás y corregir un comando erróneo sin tener que volver a escribir toda la línea. Estos comandos se resumen en la Tabla 2.12.

Tabla 2.12: Comandos del modo de control Vi para buscar en el fichero histórico

Comando	**Descripción**
k o -	Retroceder una línea.
j o +	Avanzar una línea.
G	Desplazarse a la línea indicada por la cuenta de repeticiones, o a la primera línea del historial si no hay cuenta de repeticiones.
?cadena	Buscar cadena hacia atrás.
/cadena	Buscar cadena hacia delante.
n	Repite la búsqueda en la misma dirección que la anterior.
N	Repite la búsqueda en la dirección opuesta a la anterior.

Los tres primeros pueden ir precedidos de cuentas de repetición (por ejemplo, 3k o 3- retrocede tres líneas en el archivo histórico).

Si no estás familiarizado con *vi* y su historia cultural, puede que te estés preguntando la sabiduría de elegir mnemónicos tan aparentemente pobres como `h, j, k,` y `l` para carácter de retroceso, línea de avance, línea de retroceso y carácter de avance, respectivamente. Bueno, en realidad hay una razón para las opciones - aparte de que están todos juntos en el teclado estándar.

Bill Joy desarrolló originalmente *vi* para ejecutarse en terminales Lear-Siegler ADM-3a, que fueron los primeros modelos populares con cursores direccionables (lo que significa que un programa podía enviar un comando a un ADM-3a para hacer que moviera el cursor a un lugar específico de la pantalla). Las teclas `h, j, k` y `l` de la ADM-3a tenían pequeñas flechas, así que Joy decidió usar esas teclas para los comandos apropiados en *vi*.

Otra razón (parcial) para la elección de los comandos es que CTRL-H es la tecla tradicional de retroceso, y CTRL-J denota salto de línea. La razón principal para estas elecciones, sin embargo, es que con estas teclas, nunca es necesario mover las manos de la «fila de inicio» del teclado.

Puede que + y - sean mejores mnemotécnicos que `j` y `k`, pero estos últimos tienen la ventaja de ser más accesibles para los mecanógrafos táctiles. En cualquier caso, estos comandos son los más básicos para moverse por el historial. Para ver cómo funcionan, tomemos los

mismos ejemplos que usamos al hablar del modo *emacs*.

Introduce el comando de ejemplo (ENTER funciona tanto en modo de entrada como de control, al igual que newline o CTRL-J):

```
$ fgrep -l Bob < ~pete/wk/names
```

Pero recibe un mensaje de error diciendo que la letra de su opción era incorrecta. Quieres cambiarla a *-s* sin tener que volver a escribir todo el comando. Asumiendo que estás en modo control (puede que tengas que teclear ESC para ponerte en modo control), tecleas `k` o `-` para recuperar el comando. Tu cursor estará al principio de la línea:

```
$ fgrep -l Bob < ~pete/wk/names
```

Escribe `w` para llegar al `-`, luego *l* o espacio para llegar a la `l`. Ahora puedes reemplazarlo escribiendo `rs`; presiona ENTER para ejecutar el comando.

Ahora digamos que recibes otro mensaje de error, y finalmente decides mirar la página del manual del comando `fgrep`. Recuerdas haber hecho esto hoy hace un rato, así que en lugar de teclear el comando *man(1)* entero, buscas el último que usaste. Para ello, teclea ESC para entrar en modo control (si ya estás en modo control, esto no tiene ningún efecto), luego teclea / seguido de *man* o *ma*. Para estar seguro, también puede escribir `^ma`; el ^ significa que sólo se buscarán las líneas que empiecen por `ma`.[10]

Pero teclear `/^ma` no te da lo que quieres; en su lugar, el shell te da:

```
$ make myprogram
```

Para buscar «man» de nuevo, puede escribir `n`, que realiza otra búsqueda hacia atrás utilizando la última cadena de búsqueda. Escribiendo / de nuevo sin un argumento y pulsando ENTER se consigue lo mismo.

El comando `G` recupera el comando cuyo número es el mismo que el prefijo numérico del argumento que proporcione. `G` depende del esquema de numeración de comandos descrito en la Sección 3.4.2.3, en el Capítulo 3. Sin un argumento de prefijo, va al comando número 1. Esto puede ser útil para los antiguos usuarios del intérprete de comandos C que todavía quieran utilizar números de comando.

[10] Los aficionados a *vi* y a las utilidades de búsqueda como *grep* deben tener en cuenta que el signo de intercalación (^) para el inicio de línea es el único operador contextual que el modo *vi* proporciona para las cadenas de búsqueda.

2.4.5. Comandos de Búsqueda de Caracteres

Hay algunos comandos de movimiento adicionales en el modo vi. Estos comandos le permiten moverse a la posición de un carácter particular en la línea. Se resumen en la Tabla 2.13, en la que x denota cualquier carácter.

Todos estos comandos pueden ir precedidos de una cuenta de repetición.

Tabla 2.13: Comandos de búsqueda de caracteres en modo Vi

Comando	Descripción
fx	Mover a la derecha a la siguiente ocurrencia de x (buscar).
Fx	Mover a la izquierda a la ocurrencia anterior de x (encontrar hacia atrás).
tx	Mover a la derecha a la siguiente ocurrencia de x, luego hacia atrás una posición (ir al carácter).
Tx	Mover a la izquierda a la ocurrencia anterior de x, luego hacia adelante una posición (ir hacia atrás al carácter).
;	Rehacer el último comando de búsqueda de caracteres.
,	Rehacer el último comando de búsqueda de caracteres en dirección opuesta.
%	Moverse a la coincidencia (,), {, }, [, o].

Empezando por el ejemplo anterior: supongamos que quieres cambiar *Bob* por *Rob*. Asegúrate de que estás al final de la línea (o, en cualquier caso, a la derecha de la `B` de *Bob*); entonces, si escribes `FB`, el cursor se desplaza a la `B`:

```
$ fgrep -l Bob < ~pete/wk/names
```

En este punto, podrías escribir `rR` para reemplazar la `B` por `R`. Pero digamos que quieres cambiar *Bob* por *Blob*. Necesitarías moverte un espacio a la derecha de la `B`. Por supuesto, podrías simplemente teclear `l`. Pero, dado que estás en algún lugar a la derecha de *Bob*, la forma más rápida de moverte a la o sería teclear `TB` en lugar de `FB` seguido de *l*.

Como ejemplo de cómo se puede usar la cuenta de repetición con comandos de búsqueda de caracteres, digamos que quieres cambiar el nombre del archivo de *names* a *namfile*. En este caso, asumiendo que tu cursor está todavía en la `B`, necesitas llegar a la tercera `e` a la derecha, así que puedes teclear `3te`, seguido de l para poner el cursor de nuevo en la `e` de *names*.

Los comandos de búsqueda de caracteres también tienen comandos de borrado asociados. Lee las definiciones de los comandos de la tabla anterior y sustituye mentalmente «borrar» por «mover». Obtendrá lo que ocurre cuando precede al comando de búsqueda de caracteres dado con una d. El borrado incluye el carácter dado como argumento. Por ejemplo, suponga que su cursor está bajo la n en *names*:

```
$ fgrep -l Bob < ~pete/wk/names
```

Si desea cambiar *names* por *aides*, una posibilidad es escribir `dfm`. Esto significa «borrar hasta la siguiente aparición de m», es decir, borrar «nam». A continuación, puede escribir `i` (para entrar en el modo de entrada) y luego «aid» para completar el cambio.

Una forma mejor, sin embargo, es utilizar `cfm`. Esto significa «cambiar todo desde debajo del cursor hasta la siguiente aparición de m inclusive». Esto elimina «nam» y entra en el modo de entrada.

El comando % es muy útil para encontrar el carácter «pair» coincidente cuando se utiliza con paréntesis, corchetes y llaves. Todos ellos aparecen con frecuencia en pares coincidentes en las líneas de comandos del shell.

Un último comando completa los comandos del modo de control de *vi* para desplazarse por la línea actual: puede utilizar el carácter de tubería (|) para desplazarse a una columna específica, cuyo número viene dado por un argumento de prefijo numérico. El recuento de columnas comienza en 1; cuente sólo su entrada, no el espacio ocupado por la cadena de texto. La cuenta de repetición por defecto es 1, por supuesto, lo que significa que teclear | por sí mismo equivale a 0 (ver Tabla 2.7).

2.4.6. Nombre de Archivo y Finalización y Expansión de Variables

El modo vi proporciona una característica adicional que creemos que utilizarás con bastante frecuencia: el completado de nombres de archivo. Esta característica no forma parte del editor *vi* real, y sin duda se inspiró en características similares de Emacs y, originalmente, en el sistema operativo TOPS-20 para mainframes DEC.

La lógica detrás del completado de nombres de fichero es simple: debería tener que teclear sólo lo necesario de un nombre de fichero para distinguirlo de otros nombres de fichero en el mismo directorio. La barra invertida (\) es el comando que indica al shell de Korn que complete el nombre de fichero en modo vi. Si teclea una palabra, teclea ESC para entrar en modo de control, y luego teclea \, una de cuatro cosas sucede; son las mismas que para TAB (o ESC ESC) en modo *emacs*:

1. Si no hay ningún fichero cuyo nombre empiece por la palabra, el intérprete de comandos emite un pitido y no ocurre nada más.

2. Si hay exactamente una forma de completar el nombre del fichero, y el fichero es un fichero normal, el shell teclea el resto del nombre del fichero, seguido de un espacio por si quieres teclear más argumentos de comando.

3. Si hay exactamente una forma de completar el nombre del archivo, y el archivo es un directorio, el shell completa el nombre del archivo, seguido de una barra.

4. Si hay más de una forma de completar el nombre de archivo, el shell lo completa con el prefijo común más largo entre las opciones disponibles.

Como en el modo *emacs*, a partir de *ksh93h*, puede usar TAB en lugar de ESC \. Sin embargo, esto sólo funciona si utiliza *set -o viraw* además de *set -o vi*. (La opción *viraw* consume un poco más de CPU – aunque probablemente no de forma notable – y es necesaria en algunos sistemas Unix antiguos para que el modo vi funcione). Afortunadamente, a partir de *ksh93n*, la opción *viraw* se activa automáticamente cuando se utiliza el modo vi.

Un comando relacionado es *, que es el mismo que ESC * en modo *emacs* como se describió anteriormente en este capítulo [11]. Se comporta de forma similar a ESC \, pero si hay más de una posibilidad de finalización (número cuatro en la lista anterior), enumera todas ellas y le permite seguir escribiendo. Por lo tanto, se asemeja al carácter comodín del shell *.

Por último, el comando = realiza el mismo tipo de expansión de nombre de archivo que el comodín de shell *, pero de forma diferente. En lugar de expandir los nombres de archivo en la línea de comandos, los imprime en una lista numerada con un nombre de archivo en cada línea. Luego te devuelve el prompt del shell y vuelve a escribir lo que había en tu línea de comandos antes de que teclearas =. Por ejemplo, si los archivos de tu directorio incluyen *program.c* y *problem.c*, y tecleas pro seguido de ESC y luego =, verás esto:

```
$ cc pro                          # ESC = escrito en este punto
1) problema.c
2) programa.c
$ cc pro
```

A partir de *ksh93m*, prefijar el comando = con un contador indica la selección de una opción en particular. Volviendo al ejemplo anterior: después de listar tanto *problem.c* como *program.c*, la línea de comandos queda así:

```
$ cc pro
```

[11] Si contamos la ESC necesaria para salir del modo de entrada, el comando *vi-mode* es idéntico a *emacs-mode*.

Si desea *program.c*, basta con escribir 2 =, y el intérprete de comandos elige la expansión número 2. La línea de comandos cambia a:

```
$ cc programa.c
```

Como en el modo *emacs*, también puedes completar comandos desde el modo *vi*. Los comandos *, \ y =, cuando se usan en la primera palabra de la línea de comandos, expanden alias, funciones y comandos. También como en el modo *emacs*, a partir de *ksh93l*, estas expansiones funcionan cuando has abierto una cadena entrecomillada pero aún no la has cerrado, y para expansiones de variables con `$` y `''$`.

2.4.7. Comandos Misceláneos

Varios comandos misceláneos completan el modo vi; algunos de ellos son bastante esotéricos. Se enumeran en la Tabla 2.14.

Tabla 2.14: Comandos varios del modo vi

Comando	Descripción
~	Invertir («twiddle») el caso del carácter(es) actual(es).
_	Añade la última palabra del comando anterior; entra en modo de entrada. Una cuenta repetida añade la enésima palabra dada, empezando desde el principio del comando.
v	Ejecuta el comando *hist* en la línea actual (en realidad, ejecuta el comando `hist -e ${VISUAL:-${EDITOR:-vi}}`); normalmente esto significa ejecutar el vi completo en la línea actual.
CTRL-L	Inicia una nueva línea y vuelve a dibujar la línea actual en ella; bueno para cuando tu pantalla se vuelve confusa.
CTRL-V	Imprime la versión del shell de Korn.
#	Antepone # (carácter de comentario) a la línea y la envía al fichero de historial;[12] útil para guardar un comando y ejecutarlo más tarde sin tener que volver a escribirlo. Si la línea ya empieza con #, elimina el # inicial y cualquier otro carácter de comentario que siga a nuevas líneas en un comando multilínea.
@x	Inserta la expansión del alias _x como entrada en modo comando (ver texto).

El primero de ellos puede ir precedido de una cuenta de repetición. Una cuenta de repetición de n precediendo al ~ cambia el caso de los n caracteres siguientes. [13] El cursor avanza en consecuencia.

[12] La línea también es «ejecutada» por el shell. Sin embargo, # es el carácter de comentario del shell, por lo que ésta lo ignora.

[13] Esto, en nuestra opinión, es un defecto de diseño en el editor vi que los autores del shell de Korn podrían haber corregido. Permitir al usuario añadir un comando de movimiento a y hacer que se comporte de forma análoga a d o y habría sido mucho más útil; de esa forma, se podría cambiar de mayúsculas a minúsculas una palabra con sólo dos pulsaciones de tecla.

Un conteo de repeticiones que precede a _ hace que la *n* -ésima palabra del comando anterior se inserte en la línea actual; sin el conteo, se usa la última palabra. Omitir el recuento de repeticiones es útil porque un nombre de archivo suele ser lo último en una línea de comandos de Unix y porque los usuarios suelen ejecutar varios comandos seguidos en el mismo archivo. Con esta función, puede escribir todos los comandos (excepto el primero) seguidos de ESC _ y el shell inserta el nombre del archivo.

2.4.8. Expansión de Macros con Alias

Tal como se describió anteriormente para el modo *emacs*, puede utilizar la función de alias del shell (descrita en el siguiente capítulo) para crear *macros*, es decir, abreviaturas de una sola letra para secuencias más largas de comandos. Si crea un alias llamado _x, donde x es una letra, entonces cuando escriba @ x, el modo vi expande el alias y lo lee como entrada del modo comando.

Como antes, suponga que quiere que un comando ponga en mayúscula la primera letra de la palabra actual. Podría definir un alias como sigue:

```
alias _C='B~'
```

Ahora, si escribe ESC @ C, el cursor se mueve al principio de la palabra actual (B), y luego pone en mayúscula la letra actual (~).

```
$ print here is a word  # Escribe ESC @ C
$ print here is a word
```

2.5. El comando histórico

hist es un comando incorporado en el shell [14] que proporciona un superconjunto del mecanismo de historial del shell C. Puedes usarlo para examinar los comandos más recientes que has introducido, para editar uno o más comandos con tu editor «real» favorito, y para ejecutar comandos antiguos con cambios sin tener que escribir el comando entero de nuevo. Veremos cada uno de estos usos.

La opción *-l* para *hist* lista los comandos anteriores. Toma argumentos que se refieren a órdenes en el fichero histórico. Los argumentos pueden ser números o cadenas alfanuméricas;

[14] En ksh88, este comando se llama *fc*, por «fix command». *ksh93* proporciona un alias incorporado para *fc* a *hist*, para aquellos que están acostumbrados a usar el comando *fc*. Las versiones recientes también tienen *fc* como una orden incorporada que se comporta de forma idéntica a *hist*; esto se debe a que POSIX requiere que esta orden esté incorporada.

los números se refieren a las órdenes del fichero histórico, mientras que las cadenas se refieren a la orden más reciente que empiece por la cadena. *hist* trata los argumentos de una forma bastante compleja:

- Si das dos argumentos, sirven como la primera y la última orden a mostrar.
- Si especifica un argumento numérico, sólo se muestra la orden con ese número.
- Con un único argumento de cadena, *hist* busca el comando más reciente que empiece por esa cadena y le muestra todo desde ese comando hasta el comando más reciente.
- Si no especifica ningún argumento, verá los últimos 16 comandos introducidos. Así, `hist -l` por sí mismo es equivalente al comando *history* del shell C, y de hecho el shell de Korn define un alias incorporado *history* como:

  ```
  alias history='hist -l'
  ```

  Como verá en el Capítulo 3, esto significa que puede escribir *history* y el shell de Korn ejecutará el comando `hist -l`.

Algunos ejemplos le aclararán estas opciones. Supongamos que te conectas e introduces estos comandos:

```
ls -l
more myfile
vi myfile
wc -l myfile
pr myfile | lp -h
```

Si escribe `hist -l` (o `history`) sin argumentos, verá la lista anterior con números de comando, como en:

```
1 ls -l
2 more myfile
3 vi myfile
4 wc -l myfile
5 pr mifichero | lp -h
```

La opción -n suprime los números de línea. Si quiere ver sólo los comandos 2 a 4, escriba `hist -l 2 4`. Si quiere ver sólo el comando *vi*, escriba `hist -l 3`. Para ver todo desde el comando
emphvi hasta el presente, escriba `hist -l v`. Finalmente, si desea ver los comandos entre *more* y *wc*, puede escribir `hist -l m w`, `hist -l m 4`, `hist -l 2 4`, etc.

Los números de historial negativos indican valores relativos al número de comando actual. Por ejemplo, `hist -l -3` muestra el tercer comando anterior. Una forma menos confusa

de hacer esto es con la opción *-N: hist -l -N 3* hace lo mismo. Esto también tiene la ventaja de ajustarse a las convenciones POSIX para opciones y argumentos.

La opción *-l* de *hist* no es particularmente útil, excepto como una forma rápida de recordar qué comandos ha escrito recientemente. Utilice el alias `history` si es un usuario experimentado del shell C.

La otra opción importante de *hist* es *-e* para «editar». Es útil como «escotilla de escape» de los modos *vi* y *emacs* si no estás acostumbrado a ninguno de esos editores. Puede especificar la ruta de su editor favorito y editar comandos desde su archivo de historial; entonces, cuando haya hecho los cambios, el shell ejecutará las nuevas líneas.

Digamos que tu editor favorito es una pequeña joya casera llamada *zed*. Podrías editar tus comandos escribiendo:

```
$ hist -e /usr/local/bin/zed
```

se obtiene *zed* al invocar *hist*. HISTEDIT utiliza por defecto el antiguo editor de líneas *ed*, de modo que el valor general por defecto es también *ed*.[15]

hist se utiliza normalmente para arreglar un comando reciente. Por lo tanto, maneja los argumentos de manera un poco diferente a como lo hace para la variación `hist -l` anterior:

- Sin argumentos, *hist* carga el editor con el comando más reciente.
- Con un argumento numérico, *hist* carga el editor con el comando con ese número.
- Con un argumento de cadena, *hist* carga el comando más reciente empezando por esa cadena.
- Con dos argumentos a *hist*, los argumentos especifican el principio y el final de un rango de comandos, como arriba.

Recuerde que *hist* ejecuta los comandos después de que usted los edite. Por lo tanto, la última opción puede ser peligrosa. El shell de Korn intenta ejecutar todos los comandos en el rango que especifique cuando salga de su editor. Si ha escrito alguna construcción multilínea (como las que veremos en el Capítulo 5), los resultados pueden ser incluso más peligrosos. Aunque estas pueden parecer formas válidas de generar «programas de

[15] El valor por defecto es en realidad un poco complicado en *ksh93*. `hist -e` ejecuta `${HISTEDIT:-$FCEDIT}` para editar la línea de comandos. Esto preserva la compatibilidad con *ksh88*, donde la variable para el comando *fc* era, no sorprendentemente, `FCEDIT`. Si no se establece ninguna de las dos variables, se obtiene */bin/ed*. (La construcción `${HISTEDIT:-$FCEDIT}` se explica en el Capítulo 4.) El resultado es utilizar el editor especificado por la variable `HISTEDIT` si está establecida; de lo contrario, utilice el valor de la variable `FCEDIT`).

shell instantáneos», una estrategia mucho mejor sería dirigir la salida de `hist -nl` con los mismos argumentos a un archivo; luego edite ese archivo y ejecute los comandos cuando esté satisfecho con ellos:

```
$ hist -nl cp > lastcommands        # Listar todos los comandos que empiezan por cp en
    lastcommands
$ vi lastcommands                   # Editar lastcommands
$ . lastcommands                    # Ejecuta los comandos que contiene
```

En este caso, ¡el shell no intentará ejecutar el fichero cuando abandones el editor!

Hay un último uso para *hist*. Si especifica la opción *-s* (es decir, *type hist -s*), el shell de Korn se saltará la parte de edición y sólo ejecutará el/los comando(s) especificado(s) por el/los argumento(s). ¿Por qué es útil? Por un lado, simplemente tecleando hist *-s* hace que el comando anterior se repita, igual que el comando !! del shell C. El shell de Korn proporciona el alias incorporado *r* para esto, de modo que si escribe `r` y pulsa ENTER, repetirá el último comando.

Esta forma de *hist* permite aún otro tipo de argumento, de la forma *old=new*, que significa «cambia las ocurrencias de *old* en el comando anterior especificado a *new* y luego ejecútalo». (Desafortunadamente, no puede hacer que el shell de Korn haga este tipo de sustitución más de una vez; sólo cambia la primera ocurrencia de *old* a *new*). Por ejemplo, suponga que está usando *troff* y sus preprocesadores para trabajar en un documento.[16] Si accidentalmente ejecuta el preprocesador *tbl* con este comando:

```
tbl ch2.tr | troff -ms -Tps > ch2.ps
```

pero necesitabas ejecutar *eqn*, puedes volver a hacerlo escribiendo `hist -s tbl=eqn`. (También podrías usar el alias, `r tbl=eqn`.) Este comando se ejecutaría entonces:

```
eqn ch2.tr | troff -ms -Tps > ch2.ps
```

El shell de Korn imprime el comando modificado antes de ejecutarlo.

2.6. Hábitos de los Dedos

Parafraseando el viejo adagio, los viejos hábitos de los dedos son difíciles de erradicar. De hecho, esa es la razón principal de la elección de *vi* y Emacs para los modos de edición del shell de Korn. Si eres un usuario experimentado de uno de estos editores, utiliza el modo de edición correspondiente del shell de Korn. Si eres un mago del *vi*, probablemente sepas

[16] Si es así, ¡eres de una raza rara!

cómo navegar entre dos puntos cualesquiera de una línea en tres pulsaciones de tecla o menos.

Pero si no lo eres, deberías plantearte seriamente adoptar hábitos dactilares en modo *emacs*. Dado que se basa en las teclas de control, al igual que el soporte de edición mínimo que ya habrás utilizado con el shell de Bourne o C, el modo *emacs* te resultará más fácil de asimilar. Aunque el Emacs completo es un editor extremadamente potente, su estructura de comandos se presta muy bien a pequeños subconjuntos: hay varios editores estilo «mini-emacs» flotando por ahí para Unix, MS-DOS y otros sistemas.

No se puede decir lo mismo de *vi*, porque su estructura de comandos está realmente pensada para usarse en un editor a pantalla completa. *vi* es bastante potente también, a su manera, pero su potencia se hace evidente sólo cuando se usa para propósitos similares a aquellos para los que fue diseñado: editar código fuente en C y LISP. Hacer cosas complicadas en *vi* requiere relativamente pocas pulsaciones. Pero hacer cosas sencillas requiere más pulsaciones en modo *vi* que en modo *emacs*. Por desgracia, la capacidad de hacer cosas sencillas con un número mínimo de pulsaciones es lo más deseado en un intérprete de comandos, especialmente hoy en día, cuando los usuarios pasan más tiempo dentro de las aplicaciones y menos trabajando con el intérprete de comandos.

Ambos modos de edición del intérprete de comandos Korn tienen bastantes comandos; sin duda desarrollarás hábitos de digitación que incluyen sólo algunos de ellos. Si usas el modo *emacs* y no estás familiarizado con el Emacs completo, aquí tienes un subconjunto que es fácil de aprender pero que te permite hacer casi cualquier cosa:

- Para mover el cursor por la línea de comandos, usa CTRL-A y CTRL-E para empezar y terminar la línea, y CTRL-F y CTRL-B para moverte por ella.
- Borra con DEL (o la tecla de borrado que tengas) y CTRL-D; al igual que con CTRL-F y CTRL-B, mantén pulsada la tecla para repetir si es necesario. Usa CTRL-C para borrar toda la línea.
- Utiliza CTRL-P para recuperar el último comando si cometes un error.
- Usa CTRL-R para buscar un comando que necesites ejecutar de nuevo.
- Utiliza TAB para completar nombres de archivo, comandos y variables.

Tras unas horas aprendiendo estos hábitos, te preguntarás cómo has podido vivir sin la edición en línea de comandos.

CAPÍTULO 3

PERSONALIZACIÓN DEL ENTORNO

Un sinónimo común de shell Unix, o de la interfaz que presenta cualquier programa informático, es entorno. Un *entorno* suele ser un conjunto de conceptos que expresan lo que hace un ordenador en términos diseñados para ser comprensibles y coherentes, y con un aspecto y un tacto cómodos.

Por ejemplo, tu escritorio en el trabajo es un entorno. Los conceptos relacionados con el trabajo de oficina suelen incluir notas, llamadas telefónicas, cartas, formularios, etc. Las herramientas que utilizas para ello son papel, grapas, sobres, bolígrafos, teléfono, calculadora, etc. Cada uno de estos elementos tiene una serie de características. Cada una de estas herramientas tiene una serie de características que expresan cómo las utilizas; estas características van desde la ubicación en tu escritorio o en un cajón (en el caso de las herramientas sencillas) hasta cosas más sofisticadas como a qué números están configurados los botones de memoria de tu teléfono. En conjunto, estas características conforman el aspecto de tu escritorio.

Puedes personalizar el aspecto de tu escritorio colocando los bolígrafos donde te resulte más fácil alcanzarlos, programando los botones del teléfono, etc. En general, cuanto mayor sea el grado de personalización, más adaptado estará a tus necesidades personales y, por tanto, más productivo será tu entorno.

De forma similar, los shells de Unix te presentan conceptos como ficheros, directorios y entrada y salida estándar, mientras que el propio Unix te da herramientas para trabajar con ellos, como comandos de manipulación de ficheros, editores de texto y colas de impresión. El aspecto de tu entorno Unix viene determinado por tu teclado y pantalla, por supuesto, pero también por cómo configuras tus directorios, dónde colocas cada tipo de fichero y qué nombres das a los ficheros, directorios y comandos. También hay formas más sofisticadas de personalizar tu entorno shell.

Los medios más básicos de personalización que proporciona el shell de Korn son estos:

Alias: Sinónimos de comandos o cadenas de comandos que puede definir para mayor comodidad.

Opciones: Controles para varios aspectos de tu entorno, que puedes activar y desactivar.

Variables: Marcadores de posición para información que indica al shell y a otros programas cómo comportarse en diversas circunstancias.

También hay formas más complejas de personalizar tu entorno, principalmente la capacidad de programar el shell, que veremos en capítulos posteriores. En este capítulo, cubrimos las técnicas listadas arriba.

Mientras que la mayoría de las personalizaciones obtenibles con las técnicas anteriores son sencillas y se aplican al uso diario de Unix, otras son más bien arcanas y requieren un profundo conocimiento técnico para entenderlas. La mayor parte de este capítulo se concentra en las primeras. Debido a que queremos explicar las cosas desde la perspectiva de las tareas que puede querer realizar, más que desde la de las características específicas del shell de Korn, puede que se nos escapen algunos pequeños detalles (como opciones varias para ciertos comandos). Le sugerimos que busque este tipo de información en el Apéndice B.

3.1. El Archivo .profile

Si quieres personalizar tu entorno, lo más importante es que conozcas un fichero llamado *.profile* en tu directorio home (login). Este es un archivo de comandos de shell, también llamado script de shell, que el shell de Korn lee y ejecuta cada vez que inicias sesión en tu sistema.

Si utilizas una máquina grande en una oficina o departamento, lo más probable es que el administrador del sistema ya haya configurado un fichero *.profile* para ti que contiene algunas cosas estándar. Este es uno de los ficheros «ocultos» mencionados en el Capítulo 1; otros ficheros ocultos comunes son *.xinitrc* (para el sistema X Window), *.emacs* (para el editor GNU Emacs) y *.mailrc* (para el programa de correo de Unix).

Tu *.profile*, junto con el fichero de entorno que discutiremos al final de este capítulo, será la fuente de prácticamente todas las personalizaciones que discutiremos aquí y en capítulos posteriores. Por lo tanto, es muy importante que te sientas cómodo con un editor de texto

como emphvi o Emacs para que puedas probar cualquier técnica de personalización que te apetezca.

Ten en cuenta, sin embargo, que si añades comandos a tu *.profile*, no tendrán efecto hasta que cierres la sesión y vuelvas a entrar, o escribas el comando `login`. [1] Por supuesto, no necesitas añadir inmediatamente comandos de personalización a tu .profile - siempre puedes probarlos escribiéndolos tú mismo. (Asegúrate de probar tus cambios: ¡es posible configurar cosas en tu .profile de tal manera que no puedas volver a entrar! Pruebe sus cambios antes de cerrar la sesión, iniciándola de nuevo, quizás desde una nueva ventana o consola virtual).

```
PATH=/sbin:/usr/sbin:/usr/bin:/etc:/usr/ucb:/local/bin:
stty stop ^S intr ^C erase ^?
EDITOR=/usr/local/bin/emacs
SHELL=/bin/ksh
export EDITOR
```

Estos comandos configuran un entorno básico para ti, así que probablemente no deberías cambiarlos hasta que aprendas lo que hacen – lo cual harás al final de este capítulo. Cuando edites tu *.profile*, simplemente pon tus líneas adicionales después.

3.1.1. El Archivo `/etc/profile`

Cada usuario tiene un archivo *.profile* personal en el directorio home. Aunque es posible que el administrador del sistema le haya proporcionado un archivo *.profile* inicial cuando se configuró su cuenta por primera vez, usted es libre de personalizarlo como considere oportuno.

Existe un archivo de personalización adicional, para todo el sistema, conocido como */etc/profile*. Si este archivo existe, el shell de Korn lo lee y ejecuta como la primera cosa que hace, incluso antes de leer su archivo personal *.profile*. Aquí es donde el administrador del sistema coloca los comandos que deben ser ejecutados por cada usuario al iniciar sesión, y donde él o ella coloca los valores por defecto de todo el sistema, tales como añadir directorios adicionales a la variable `PATH` (que, como verá más adelante en este capítulo, le dice al shell dónde buscar programas para ejecutar).

Vale la pena estar al tanto de este archivo, ya que puede contener configuraciones que usted podría desear anular en su propio archivo *.profile*. (Si el archivo existe, será legible

[1] Esto tiene el mismo efecto que cerrar la sesión y volver a iniciarla, aunque en realidad sustituye tu sesión de inicio de sesión por una nueva sin terminar explícitamente la sesión anterior.

y contendrá comandos de shell, al igual que su *.profile*. Puede valer la pena examinar la versión en su sistema; puede aprender algo de esa manera.

3.2. Alias

Quizás el tipo de personalización más fácil y popular es el *alias*, que es un sinónimo de un comando o cadena de comandos. Esta es una de varias características del shell de Korn que fueron apropiadas del shell C.[2] Usted define un alias introduciendo (o añadiendo a su *.profile*) una línea con la siguiente forma:

```
alias new=original
```

(Observe que no hay espacios a ambos lados del signo igual (=); es la sintaxis requerida). El comando *alias* define *new* como un alias de *original*; cada vez que escriba *new*, el shell de Korn sustituirá internamente a *original*. (No puede utilizar ninguno de los caracteres especiales del shell, como *, $, =, etc., en los nombres de alias).

Existen varias formas básicas de utilizar un alias. La primera, y más sencilla, es utilizar un nombre más mnemotécnico para una orden existente. Muchos comandos Unix de uso común tienen nombres que son mnemotécnicos pobres y por lo tanto son excelentes candidatos para alias; el ejemplo clásico es:

```
alias search=grep
```

grep, la utilidad de búsqueda de archivos de Unix, deriva su nombre del comando «g/re/p» del editor de texto *ed* original, que hace esencialmente lo mismo que *grep*. (El código de búsqueda de expresiones regulares se eliminó de *ed* para crear un programa independiente).[3] Este acrónimo puede significar algo para un informático, pero probablemente no para el administrador de una oficina que tiene que encontrar a `Fred` en una lista de números de teléfono. Si tiene que encontrar a `Fred`, y tiene la palabra *search* definida como alias de *grep*, puede escribir:

```
search Fred phonelist
```

Otro alias popular evita *exit* en favor de un comando más utilizado para finalizar una sesión de inicio de sesión:

```
alias logout=exit
```

[2] Los usuarios del shell C deberían tener en cuenta que la característica alias del shell de Korn no soporta argumentos en las expansiones de alias, como hacen los alias del shell C.

[3] Gracias a Dennis Ritchie y Brian Kernighan, de los laboratorios Bell, por verificarlo. ADR.

Si eres un usuario del shell C, puede que estés acostumbrado a tener un fichero *.logout* de comandos que el shell ejecuta justo antes de cerrar la sesión. El shell de Korn no tiene esta característica como tal, pero puedes imitarla fácilmente usando un alias:

```
alias logout='. ~/.ksh\_logout; exit'
```

Esto ejecuta los comandos en el archivo *.ksh_logout* en su directorio home y luego cierra su sesión. El punto y coma actúa como separador de sentencias, permitiéndote tener más de un comando en la misma línea.

Fíjese en las comillas que rodean el valor completo del alias; son necesarias si la cadena a la que se asigna el alias consta de más de una palabra.[4]

Es posible que desee el archivo *.ksh_logout* para «limpiar» sus archivos de historia, como discutimos en el último capítulo. Recuerde que creamos archivos de historial con nombres como *.hist.42*, lo que garantiza un nombre único para cada línea serie o ventana. Para eliminar estos archivos cuando los shells salgan, simplemente ponga esta línea en su archivo *.ksh_logout*:

```
rm ~/.hist.*
```

A algunas personas que no son especialmente buenas mecanógrafas les gusta utilizar alias para los errores tipográficos que cometen a menudo. Por ejemplo:

```
alias emcas=emacs
alias mali=mail
alias gerp=grep
```

Esto puede ser útil, pero creemos que probablemente es mejor sufrir con el mensaje de error y tener la ortografía correcta bajo los dedos. Otra forma habitual de utilizar un alias es como abreviatura de una cadena de comandos más larga. Por ejemplo, puede que tengas un directorio al que necesites ir a menudo. Está enterrado profundamente en tu jerarquía de directorios, así que quieres establecer un alias que te permita ir allí sin teclear (o incluso recordar) la ruta completa:

```
alias cdcm='cd ~/work/projects/devtools/windows/confman'
```

Como antes, las comillas alrededor del comando *cd* completo son necesarias, porque la cadena que se está aliaseando tiene más de una palabra.

Por ejemplo, una opción útil del comando *ls* es *-F*: pone una barra (/) después de los archivos de directorio y un asterisco (*) después de los archivos ejecutables. (Dependiendo

[4] [37] Esto contrasta con los alias del shell C, en los que las comillas no son necesarias.

de tu sistema, también puede añadir otros caracteres después de otros tipos de archivos). Como escribir un guión seguido de una letra mayúscula es incómodo, a mucha gente le gusta definir un alias de esta manera:

```
alias lf='ls -F'
```

Es importante recordar algunas cosas sobre los alias. En primer lugar, el shell de Korn hace una sustitución textual del alias por aquello a lo que está poniendo el alias; puede ayudar imaginarse a *ksh* pasando su comando a través de un editor de texto o procesador de textos y emitiendo un comando «change» o «substitute» antes de interpretarlo y ejecutarlo.

Esto, a su vez, significa que cualquier carácter especial (como comodines como * y ?) que resulten cuando se expande el alias son interpretados correctamente por el shell. Esto lleva a un corolario importante: los comodines y otros caracteres especiales no pueden utilizarse en los nombres de los alias, es decir, a la izquierda del signo igual. Por ejemplo, para facilitar la impresión de todos los archivos de su directorio, podría definir el alias:

```
alias printall='pr * | lp'
```

En segundo lugar, ten en cuenta que los alias son recursivos, lo que significa que es posible poner un alias a otro alias. Una objeción legítima al ejemplo anterior es que el alias, aunque mnemotécnico, es demasiado largo y no ahorra teclear lo suficiente. Si queremos mantener este alias pero añadir una abreviatura más corta, podríamos definir:

```
 alias pa=printall
```

El alias recursivo hace posible establecer un «bucle infinito» de definiciones, en el que un alias termina (quizás después de varias búsquedas) siendo definido como sí mismo. Por ejemplo, el comando:

```
alias ls='ls -l'
```

crea un posible bucle infinito. Por suerte, el shell tiene un mecanismo para protegerse de estos peligros. El comando anterior funciona como se espera (al teclear *ls* se obtiene una larga lista con permisos, tamaños, propietarios, etc.). Incluso funcionan situaciones más patológicas, como éstas:

```
alias listfile=ls
alias ls=listfile
```

Si escribes *listfile*, ls se ejecuta.

Los alias sólo pueden utilizarse al principio de una cadena de comandos, aunque con algunas excepciones. En el ejemplo anterior de *cd*, es posible que desee definir un alias sólo para el

nombre del directorio, no para todo el comando. Pero si define:

```
alias cm=work/projects/devtools/windows/confman
```

y luego escriba `cd cm`, el shell de Korn probablemente imprimirá un mensaje como `ksh: cd: cm: [No such file or directory]`.

Una característica oscura y bastante fea de la facilidad de alias del shell de Korn – que no está presente en la característica análoga del shell C – proporciona una forma de evitar este problema. Si el valor de un alias (el lado derecho del signo igual) termina en un espacio o en un tabulador, entonces el shell de Korn intenta hacer la sustitución del alias en la siguiente palabra de la línea de órdenes. Para hacer que el valor de un alias termine en un espacio, debe rodearlo con comillas.

Esta característica existe para que sea posible tener alias para comandos que a su vez ejecutan otros comandos, como *nohup* y *nice*. Por ejemplo, *nohup* tiene el alias `'nohup '`. De esta forma, cuando escribes

```
nohup my_favorite_alias somefile
```

el shell expandirá `my_favorite_alias` tal como lo haría cuando se escribe sin el comando *nohup* precedente. (El comando *nohup* se describe en el Capítulo 8.)

A continuación se muestra cómo utilizar esta capacidad para permitir alias para nombres de directorio, al menos para su uso con el comando *cd*:

```
alias cd='cd '
```

Esto hace que el shell de Korn busque un alias para el argumento de nombre de directorio a *cd*, que en el ejemplo anterior le permitiría expandir el alias *cm* correctamente.

El shell de Korn proporciona una característica de eficiencia llamada «alias rastreados». Retrasaremos la discusión de estos hasta la Sección 3.4.2.8. Además, varios alias están predefinidos por el shell; están listados en el Apéndice B.

Por último, existen algunos complementos útiles para el comando *alias* básico. Si escribe `alias name` sin un signo igual (=) y value, el shell imprime el valor del alias o *name:* `alias not found` si no está definido. Si escribe *alias* sin ningún argumento, obtendrá una lista de todos los alias que haya definido, así como de varios que están incorporados. Si escribe `alias -p`, el shell imprime todos sus alias, con cada uno precedido por la palabra clave *alias*. Esto es útil para guardar todos sus alias de forma que el shell pueda volver a leerlos

en otro momento. El comando `unalias` *name* elimina cualquier definición de alias para su argumento. Si escribe `unalias -a`, el shell elimina todos los alias.

Los alias son muy útiles para crear un entorno cómodo, pero en realidad son sólo cosas de niños en comparación con técnicas de personalización más avanzadas como scripts y funciones, que veremos en el próximo capítulo. Éstas te dan todo lo que hacen los alias y mucho más, así que si te vuelves experto en ellas, puede que descubras que ya no necesitas los alias. Sin embargo, los alias son ideales para los novatos que encuentran a Unix como un lugar bastante prohibitivo, lleno de tersura y desprovisto de buenos mnemotécnicos.

3.3. Opciones

Aunque los alias le permiten crear nombres convenientes para los comandos, en realidad no le permiten cambiar el comportamiento del shell. Las *opciones* son una forma de hacerlo. Una opción del shell es un ajuste que puede estar «activado» o «desactivado». Mientras que varias opciones se relacionan con características arcanas del shell que son de interés sólo para los programadores, las que cubrimos aquí son de interés para los programadores.

Los comandos básicos relacionados con las opciones son *set -o optionnames* y *set +o optionnames*, donde *optionnames* es una lista de nombres de opciones separados por espacios en blanco. El uso de los signos más (+) y menos (-) es contradictorio: el - activa la opción nombrada, mientras que el + la desactiva. La razón de esta incongruencia es que el guión (-) es la forma convencional de Unix de especificar opciones a un comando, mientras que el uso de + es una ocurrencia posterior.

La mayoría de las opciones también tienen abreviaturas de una letra que pueden utilizarse en lugar del comando *set -o*; por ejemplo, *set -o noglob* puede abreviarse *set -f*. Estas abreviaturas provienen del shell de Bourne. Al igual que otras características «extra» del shell de Korn, existen para asegurar la compatibilidad; de lo contrario, no se recomienda su uso.

La Tabla 3.1 lista las opciones que son útiles para los usuarios generales de Unix. Todas ellas están desactivadas por defecto excepto cuando se indica lo contrario.

Tabla 3.1: Opciones básicas del shell

Opción	Descripción
bgnice	Ejecutar trabajos en segundo plano con prioridad baja (activada por defecto).
emacs	Entra en el modo de edición emacs.
ignoreeof	No permitir el uso de CTRL-D para cerrar la sesión; requerir el comando exit.
markdirs	Al expandir los comodines de nombre de archivo, añada una barra (/) a los directorios.
noclobber	No permita que la redirección de salida (>) destruya un archivo existente.
noglob	No expanda los comodines de nombre de archivo como * y ? (la expansión de comodines a veces se denomina *globbing*).
nounset	Indicar un error al intentar utilizar una variable que no está definida.
trackall	Activar el seguimiento de alias. (En realidad, el shell ignora la configuración de esta opción; el seguimiento de alias siempre está activado. Esto se discute en la Sección 3.4.2.8, más adelante en este capítulo).
vi	Entra en el modo de edición vi.

Existen otras opciones (22 en total; el Apéndice B las enumera). Para comprobar el estado de una opción, escriba *set -o*. El shell de Korn imprime una lista de todas las opciones junto con su configuración. No existe un comando para probar opciones individuales, pero aquí hay una simple función del shell para hacerlo:

```
function testopt {
  if [[ -o $1 ]] ; then
    print Option $1 is on.
  else
    print Option $1 is off.
  fi
}
```

Las funciones de shell se tratan en el próximo capítulo. Por ahora, sin embargo, si desea utilizar la función *testopt*, simplemente escríbala en su archivo *.profile* o de entorno (consulte la Sección 3.5.2, más adelante en este capítulo), escriba `login` o `. .profile`. (Sí, el punto, o «dot», es en realidad un comando; vea la Sección 4.1 en el Capítulo 4.) Luego puede escribir *testopt optionname* para comprobar el estado de una opción.

3.4. Variables de Shell

Hay varias características de su entorno que puede que desee personalizar pero que no pueden expresarse como una opción on/off. Las características de este tipo se especifican

en variables de shell. Las variables de shell pueden especificarlo todo, desde la cadena del prompt hasta la frecuencia con la que el shell comprueba si hay correo nuevo.

Al igual que un alias, una variable de shell es un nombre que tiene un valor asociado. El shell de Korn mantiene un registro de varias variables del shell incorporadas; los programadores del shell pueden añadir las suyas propias. Por convención, las variables incorporadas tienen nombres en mayúsculas. La sintaxis para definir variables es similar a la sintaxis para alias:

```
varname=value
```

No debe haber ningún espacio a ambos lados del signo igual, y si el valor tiene más de una palabra, debe ir entre comillas. Para utilizar el valor de una variable en un comando, preceda su nombre con un signo de dólar ($).

Puede eliminar una variable con el comando `unset varname`. Normalmente, esto no es útil, ya que se supone que todas las variables que no existen son nulas, es decir, iguales a la cadena vacía "". Pero si utiliza la opción `nounset` (véase la Tabla 3.1), que hace que el shell indique un error cuando encuentra una variable indefinida, puede interesarle unset.

La forma más sencilla de comprobar el valor de una variable es utilizar el comando integrado `print`[5]. Todo lo que hace `print` es imprimir sus argumentos, pero no hasta que el shell los haya evaluado. Esto incluye – entre otras cosas que se discutirán más adelante – tomar los valores de las variables y expandir los comodines de nombre de archivo. Así, si la variable *fred* tiene el valor `bob`, escribir lo siguiente hace que el shell simplemente imprima `bob`:

```
print "$fred"
```

Si la variable no está definida, el shell imprime una línea en blanco. Una forma más verbosa de hacer esto es:

```
print "The value of \$varname is \"$varname\"."
```

El primer signo de dólar y las comillas dobles interiores se escriben con barra diagonal inversa (es decir, precedidas de para que el shell no intente interpretarlas; véase el Capítulo 1) para que aparezcan literalmente en la salida, que para el ejemplo anterior sería:

```
The value of $fred is "bob".
```

[5]El shell de Korn soporta el antiguo comando `echo`, que hace prácticamente lo mismo, por razones de compatibilidad con versiones anteriores. Sin embargo, recomendamos encarecidamente `print` porque sus opciones son las mismas en todos los sistemas Unix, mientras que las opciones de `echo` difieren entre las diferentes versiones de Unix. No es probable que esto cambie; el estándar POSIX dice que las opciones de `echo` están definidas por la implementación.

3.4.1. Variables y Citas

Fíjese en que hemos utilizado comillas dobles alrededor de las variables (y de las cadenas que las contienen) en estos ejemplos de *impresión*. En el Capítulo 1 dijimos que algunos caracteres especiales entre comillas dobles se interpretan (mientras que ninguno se interpreta entre comillas simples).

Quizás el carácter especial más importante que «sobrevive» a las comillas dobles es el signo del dólar, que significa que las variables se evalúan. Es posible prescindir de las comillas dobles en algunos casos; por ejemplo, podríamos haber escrito la orden *print* anterior de esta forma:

```
print The value of \$varname is \"$varname\".
```

Pero las comillas dobles son más generalmente correctas.

He aquí por qué. Supongamos que hacemos esto:

```
fred='Four spaces between these words.'
```

Entonces, si introducimos el comando print `$fred`, el resultado sería:

```
Four spaces between these words.
```

¿Qué ha pasado con los espacios extra? Sin las comillas dobles, el shell divide la cadena en palabras después de sustituir el valor de la variable, como hace normalmente cuando procesa líneas de comandos. Las comillas dobles evitan esta parte del proceso (haciendo que el shell piense que toda la cadena entrecomillada es una sola palabra).

Por lo tanto el comando `print "$fred"` imprime esto:

```
Four spaces between these words.
```

Esto se vuelve especialmente importante cuando empezamos a tratar con variables que contienen entradas de usuario o archivos más adelante. En particular, cada vez es más común encontrar directorios disponibles en sistemas Unix a través de la red desde sistemas Apple Macintosh y Microsoft Windows, donde los espacios y otros caracteres inusuales son comunes en los nombres de archivos.

Las comillas dobles también permiten que funcionen otros caracteres especiales, como veremos en el Capítulo 4, el Capítulo 6 y el Capítulo 7. Pero por ahora, revisaremos la regla «En caso de duda, use comillas simples» del capítulo 1 añadiendo: "...a menos que una cadena contenga una variable, en cuyo caso deberá usar comillas dobles».

3.4.2. Variables Incorporadas

Al igual que con las opciones, algunas variables incorporadas al shell son significativas para los usuarios generales de Unix, mientras que otras son arcanos para los programadores profesionales. Aquí veremos las más útiles en general, y dejaremos algunas de las más oscuras para capítulos posteriores. De nuevo, el Apéndice B contiene una lista completa.

Edición de Variables de Modo

Varias variables del shell se relacionan con los modos de edición de la línea de comandos que vimos en el capítulo anterior. Éstas se enumeran en la Tabla 3.2.

Las dos primeras son usadas a veces por editores de texto y otros programas orientados a pantalla, que dependen de que las variables estén configuradas correctamente. Aunque el shell de Korn y la mayoría de los sistemas de ventanas deberían saber cómo establecerlas correctamente, debería mirar los valores de `COLUMNS` y `LINES` si tiene problemas de visualización con un programa orientado a pantalla.

Tabla 3.2: Variables del modo de edición

Variable	Significado
`COLUMNS`	Anchura, en columnas de caracteres, de su terminal. El valor estándar es 80 (a veces 132), aunque si está utilizando un sistema de ventanas como X, puede dar a una ventana de terminal cualquier tamaño que desee.
`LINES`	Longitud de su terminal en líneas de texto. El valor estándar para terminales es 24, pero para monitores compatibles con IBM PC es 25; una vez más, si está utilizando un sistema de ventanas, normalmente puede cambiar el tamaño a cualquier cantidad.
`HISTFILE`	Nombre del fichero histórico sobre el que operan los modos de edición.
`EDITOR`	Nombre de ruta de tu editor de texto favorito; el sufijo (`macs`[6] o `vi`) determina qué modo de edición utilizar.
`VISUAL`	Similar a `EDITOR`; si se establece, se utiliza con preferencia a `EDITOR` para elegir el modo de edición.
`HISTEDIT`	Nombre de ruta del editor que se utilizará con el comando *hist*.

Variables de Correo

Dado que el programa de correo no se ejecuta todo el tiempo, no hay forma de que le informe cuando recibe correo nuevo; por lo tanto, el intérprete de órdenes hace esto en su lugar[7].

[6] Este sufijo también funciona si tu editor es una versión diferente de Emacs cuyo nombre no termina en emacs.

[7] El comúnmente disponible comando *biff* hace un mejor trabajo en este sentido; mientras que el shell de Korn sólo imprime mensajes de «tiene correo» justo antes de imprimir las indicaciones de comandos,

El intérprete de órdenes no puede comprobar realmente si hay correo entrante, pero puede mirar su fichero de correo periódicamente y determinar si el fichero ha sido modificado desde la última comprobación. Las variables listadas en la Tabla 3.3 le permiten controlar cómo funciona esto.

Tabla 3.3: Variables de correo

Variable	**Significado**
`MAIL`	Nombre del archivo para comprobar el correo entrante (es decir, su archivo de correo)
`MAILCHECK`	Con qué frecuencia, en segundos, se comprueba si hay correo nuevo (por defecto 600 segundos, o 10 minutos)
`MAILPATH`	Lista de nombres de archivo, separados por dos puntos (:), para comprobar el correo entrante
`_(guión bajo)`	Cuando se utiliza dentro de `$MAILPATH`, nombre del archivo de correo que ha cambiado; véase el texto para otros usos

En el escenario más simple, usted utiliza el programa de correo estándar de Unix, y su fichero de correo es */var/mail/yourname* o algo similar. En este caso, sólo tendría que establecer la variable `MAIL` a este nombre de archivo si desea que se compruebe su correo:

```
MAIL=/var/mail/yourname
```

Si su administrador del sistema no lo ha hecho ya por usted, ponga una línea como ésta en su *.profile*.

Sin embargo, algunas personas utilizan *mailers* no estándar que usan múltiples ficheros de correo; `MAILPATH` fue diseñado para acomodar esto. El shell de Korn utiliza el valor de `MAIL` como el nombre del fichero a comprobar, a menos que se establezca `MAILPATH`, en cuyo caso el shell comprueba cada fichero de la lista `MAILPATH` en busca de correo nuevo. Puede utilizar este mecanismo para que el shell imprima un mensaje diferente para cada archivo de correo: para cada nombre de archivo de correo en `MAILPATH`, añada un signo de interrogación seguido del mensaje que desea imprimir.

Por ejemplo, supongamos que tienes un sistema de correo que ordena automáticamente el correo en archivos según el nombre de usuario del remitente. Tiene archivos de correo llamados */var/mail/usted/fritchie*, */var/mail/usted/droberts*, */var/mail/usted/jphelps*, etc. Usted define su `MAILPATH` como sigue:

```
MAILPATH=/var/mail/you/fritchie:/var/mail/you/droberts:\
/var/mail/you/jphelps
```

biff puede decirle de quién es el correo.

Si recibe correo de Jennifer Phelps, el archivo */var/mail/you/jphelps* cambia. El shell de Korn se da cuenta del cambio en 10 minutos e imprime el mensaje:

```
you have mail in /var/mail/you/jphelps.
```

Si está en medio de la ejecución de un comando, el shell espera hasta que el comando finalice (o se suspenda) para imprimir el mensaje. Para personalizar esto aún más, puede definir `MAILPATH` como:

```
MAILPATH=\
/var/mail/you/fritchie?You have mail from Fiona.:\
/var/mail/you/droberts?Mail from Dave has arrived.:\
/var/mail/you/jphelps?There is new mail from Jennifer.
```

Las barras invertidas al final de cada línea le permiten continuar su comando en la línea siguiente. Pero ten cuidado: no puedes sangrar las líneas siguientes. Ahora, si recibe correo de Jennifer, el shell imprime:

```
There is new mail from Jennifer.
```

Dentro de las partes de mensaje de `MAILPATH`, puede utilizar la variable especial _ (guión bajo) para el nombre del archivo que activa el mensaje:

```
MAILPATH='/var/mail/you/fritchie?You have mail from Fiona in \$_.'
MAILPATH+=':/var/mail/you/droberts?Mail from Dave has arrived, check \$_.'
MAILPATH+=':/var/mail/you/jphelps?There is new mail from Jennifer, look at \$_.'
```

En realidad, el significado de $_ varía en función de dónde y cómo se utilice:

Inside the value of MAILPATH: Como se acaba de describir, utilice $_ para el nombre del archivo que activa un mensaje en el valor de MAILPATH.

El último argumento del último comando interactivo: Cuando se utiliza en una línea de comandos introducida de forma interactiva, $_ representa la última palabra de la línea de comandos anterior:

```
$ print hi    # Ejecutar un comando
hi
$ print _$    # Verificar la configuracion de $_
hi
$ print hello   # Nuevo ultimo argumento
hello
$ print _$
hello
$ print "hi there"  # El uso se basa en palabras
hi there
$ print _$
hi there
```

Este uso de $_ es similar a la función !$ del mecanismo de historial del intérprete de comandos C.

Dentro de un script: Cuando se accede desde dentro de un script de shell, $_ es la ruta completa utilizada para encontrar e invocar el script:

```
$ cat /tmp/junk   # Mostrar programa de pruebas
print _ is $_
$ PATH=/tmp:$PATH # Agregar directorio a PATH
$ junk      # Ejecutar el programa
_ is /tmp/junk
```

Variables de prompt

Si has visto trabajar a suficientes usuarios experimentados de Unix, puede que ya te hayas dado cuenta de que el prompt del shell no está grabado en piedra. Parece que uno de los pasatiempos favoritos de los programadores profesionales de Unix es pensar en cadenas de prompt bonitas o innovadoras. Te daremos algo de la información que necesitas para hacer la tuya aquí; el resto viene en el próximo capítulo.

En realidad, el shell de Korn utiliza cuatro cadenas de prompt. Se almacenan en las variables `PS1`, `PS2`, `PS3` y `PS4`. La primera de ellas se llama prompt primario; es el prompt habitual del shell, y su valor por defecto es `"$ "` (un signo de dólar seguido de un espacio). A mucha gente le gusta establecer su prompt primario a algo que contenga su nombre de usuario. Esta es una forma de hacerlo:

```
PS1="($LOGNAME)-> "
```

`LOGNAME` es otra variable incorporada en el shell, que se establece con tu nombre de usuario cuando te conectas[8]. Así, `PS1` se convierte en un paréntesis a la izquierda, seguido de tu nombre de usuario, seguido de `")-> "`. Si su nombre de usuario es fred, su prompt será `"(fred)-> "`.Si usted es un usuario del shell C y, como muchas de esas personas, está acostumbrado a tener un número de comando en su cadena del prompt, el shell de Korn puede hacer esto de forma similar al shell C: si hay un signo de exclamación en la cadena del prompt, sustituye el número de comando. Así, si define su cadena de comandos como la siguiente, sus comandos se verán como `(fred 1)->`, `(fred 2)->` , y así sucesivamente:

```
PS1="($LOGNAME !)->"
```

Quizás la forma más útil de configurar tu cadena de prompt es que siempre contenga tu directorio actual. Así no necesitará teclear `pwd` para recordar dónde se encuentra. Poner su

[8] Algunos sistemas muy antiguos utilizan `USER` en su lugar. Afortunadamente, estos sistemas son cada vez más raros.

directorio en el prompt es más complicado que los ejemplos anteriores, porque su directorio actual cambia durante su sesión de login, a diferencia de su nombre de usuario y el nombre de su máquina. Pero podemos acomodar esto aprovechando los diferentes tipos de comillas. He aquí cómo:

```
PS1='($PWD)-> '
```

La diferencia son las comillas simples, en lugar de las dobles, que rodean la cadena a la derecha de la asignación. El truco está en que esta cadena se evalúa dos veces: una cuando se realiza la asignación a `PS1` (en su archivo *.profile* o de entorno) y otra después de cada comando que introduzca. Esto es lo que hace cada una de estas evaluaciones:

1. La primera evaluación observa las comillas simples y devuelve lo que hay dentro de ellas sin más procesamiento. Como resultado, `PS1` contiene la cadena `($PWD)->` .
2. Después de cada comando, el shell evalúa `($PWD)->`. `PWD` es una variable incorporada que siempre es igual al directorio actual, por lo que el resultado es un prompt primario que siempre contiene el directorio actual[9].

En el capítulo 7 analizaremos más a fondo las sutilezas de la cita y la evaluación diferida.

`PS2` se denomina cadena secundaria; su valor por defecto es "> " (un signo mayor que seguido de un espacio). Se utiliza cuando escribe una línea incompleta y pulsa ENTER, como indicación de que debe terminar su comando. Por ejemplo, suponga que comienza una cadena entrecomillada pero no cierra las comillas. Si pulsa INTRO, el shell imprime > y espera a que termine la cadena:

```
$ x="This is a long line,   # PS1 para los comandos
> which is terminated down here"  # PS2 para la continuacion
$         # PS1 para el siguiente comando
```

`PS3` y `PS4` se refieren a la programación y depuración del shell, respectivamente; se explican en el Capítulo 5 y el Capítulo 9.

Uso de los Números de Comando del Historial

El número del comando histórico actual está disponible en la variable de entorno `HISTCMD`. Puede ver el número de historial actual en su prompt colocando un ! (o `$HISTCMD`) en algún lugar del valor de la variable `PS1`:

[9]El intérprete de comandos también realiza sustituciones aritméticas y de comandos en el valor de `PS1`, pero aún no hemos tratado estas funciones. Véase el Capítulo 6.

```
$ PS1="command !> "
command 42> ls -FC *.xml
appa.xml appd.xml ch01.xml ch04.xml ch07.xml ch10.xml
appb.xml appf.xml ch02.xml ch05.xml ch08.xml colo1.xml
appc.xml ch00.xml ch03.xml ch06.xml ch09.xml copy.xml
command 43>
```

Para obtener un literal ! en el valor de su prompt, coloque !! en `PS1`.

Tipo de Terminales

Hoy en día, el uso más común del intérprete de comandos es desde dentro de una ventana de emulador de terminal que se muestra en la pantalla de alta resolución de una estación de trabajo o PC. Sin embargo, el programa emulador de terminal todavía emula las facilidades proporcionadas por los terminales CRT serie reales de antaño. Como tal, la variable de shell `TERM` es de vital importancia para cualquier programa que utilice toda su ventana, como un editor de texto. Tales programas incluyen editores de pantalla tradicionales (como *vi* y Emacs), programas de paginación como *more*, e innumerables aplicaciones de terceros.

Debido a que los usuarios pasan cada vez más tiempo dentro de los programas y cada vez menos utilizando el propio shell, es extremadamente importante que su `TERM` esté configurado correctamente. Es realmente el trabajo de su administrador del sistema para ayudarle a hacer esto (o hacerlo por usted), pero en caso de que necesite hacerlo usted mismo, aquí hay algunas pautas.

El valor de `TERM` debe ser una cadena de caracteres corta con letras minúsculas que aparezca como nombre de archivo en la base de datos *terminfo*.[10] Esta base de datos es un directorio de dos niveles de archivos bajo el directorio raíz */usr/share/terminfo*.[11] Este directorio contiene subdirectorios con nombres de un solo carácter; éstos, a su vez, contienen archivos de información del terminal para todos los terminales cuyos nombres empiecen por ese carácter. Cada archivo describe cómo decirle al terminal en cuestión que haga ciertas cosas comunes como colocar el cursor en la pantalla, pasar a vídeo inverso, desplazarse, insertar texto, etc. Las descripciones están en formato binario (es decir, no legibles por humanos).

Los nombres de los archivos de descripción de terminales son los mismos que los del ter-

[10] Las versiones de Unix no derivadas de System V utilizan *termcap*, una base de datos de capacidades de terminal de estilo antiguo que utiliza el único archivo de texto */etc/termcap* para todas las descripciones de terminal. Los sistemas modernos suelen tener disponibles tanto el fichero */etc/termcap* como la base de datos *terminfo*. Los sistemas BSD actuales usan una base de datos indexada de un solo fichero, */usr/share/misc/termcap.db*.

[11] Esta es la ubicación típica en los sistemas modernos. Los sistemas más antiguos lo tienen en */usr/lib/terminfo*.

minal que se está describiendo; a veces se utiliza una abreviatura. Por ejemplo, el DEC VT100 tiene una descripción en el archivo */usr/share/terminfo/v/vt100*; la consola basada en caracteres GNU/Linux tiene una descripción en el archivo */usr/share/terminfo/l/linux*. Una ventana de terminal xterm bajo el sistema X Window tiene una descripción en */usr/share/terminfo/x/xterm*.

A veces su software Unix no configura `TERM` correctamente; esto ocurre a menudo en terminales X y sistemas Unix basados en PC. Por lo tanto, debería comprobar el valor de `TERM` tecleando `print $TERM` antes de seguir adelante. Si encuentra que su sistema Unix no está configurando el valor correcto para usted (especialmente probable si su terminal es de una marca diferente a la de su ordenador), necesitará encontrar el valor apropiado de `TERM` usted mismo.

La mejor manera de encontrar el valor `TERM` – si no puede encontrar un gurú local que lo haga por usted – es adivinar el nombre *terminfo* y buscar un archivo con ese nombre bajo */usr/share/terminfo* usando *ls*. Por ejemplo, si tu terminal es un Blivitz BL-35A, podrías probar:

```
$ cd /usr/share/terminfo
$ ls b/bl*
```

Si tiene éxito, verá algo como esto:

```
bl35a   blivitz35a
```

En este caso, es probable que los dos nombres sean sinónimos de (enlaces a) la misma descripción de terminal, por lo que podría utilizar cualquiera de ellos como valor de `TERM`. En otras palabras, podría poner cualquiera de estas dos líneas en su *.profile*:

```
TERM=bl35a
TERM=blivitz35a
```

Si no tiene éxito, *ls* no imprimirá nada, y tendrá que hacer otra suposición e intentarlo de nuevo. Si encuentra que *terminfo* no contiene nada que se parezca a su terminal, no todo está perdido. Consulte el manual de su terminal para ver si el terminal puede emular un modelo más popular; hoy en día las probabilidades de que esto ocurra son excelentes.

Por el contrario, *terminfo* puede tener varias entradas relacionadas con su terminal, para submodelos, modos especiales, etc. Si puede elegir qué entrada utilizar como valor de `TERM`, le sugerimos que pruebe cada una de ellas con su editor de texto o cualquier otro programa orientado a pantalla que utilice y vea cuál funciona mejor.

El proceso es mucho más sencillo si estás utilizando un sistema de ventanas, en el que tus «terminales» son porciones lógicas de la pantalla en lugar de dispositivos físicos. En este caso, el software dependiente del sistema operativo fue escrito para controlar su(s) ventana(s) de terminal, por lo que las probabilidades son muy buenas de que si sabe cómo manejar el cambio de tamaño de la ventana y el complejo movimiento del cursor, sea capaz de lidiar con cosas simples como `TERM`. El sistema X Window, por ejemplo, establece automáticamente «xterm» como valor para `TERM` en una ventana de terminal *xterm*.

Ruta de Búsqueda de Comandos

Otra variable importante es `PATH`, que ayuda al shell a encontrar los comandos que introduzcas.

Como probablemente sepas, cada comando que utilizas es en realidad un archivo que contiene código para que tu máquina lo ejecute.[12] Estos archivos se denominan archivos ejecutables o simplemente ejecutables. Se almacenan en varios directorios. Algunos directorios, como */bin* o */usr/bin*, son estándar en todos los sistemas Unix; otros dependen de la versión concreta de Unix que estés utilizando; algunos son exclusivos de tu máquina; si eres programador, algunos pueden ser incluso tuyos. En cualquier caso, no hay ninguna razón por la que debas saber dónde está el archivo ejecutable de un comando para poder ejecutarlo.

Aquí es donde entra `PATH`. Su valor es una lista de directorios que el shell busca cada vez que usted introduce un nombre de comando que no contiene una barra; los nombres de directorio están separados por dos puntos (:), igual que los archivos en `MAILPATH`. Por ejemplo, si escribe `print $PATH`, verá algo como esto:

```
/sbin:/usr/sbin:/usr/bin:/etc:/usr/X11R6/bin:/local/bin
```

¿Por qué debería preocuparse por su path? Hay tres razones principales. Primero, hay aspectos de seguridad en su valor, que tocaremos en breve. En segundo lugar, una vez que haya leído los últimos capítulos de este libro e intente escribir sus propios programas shell, querrá probarlos y eventualmente reservar un directorio para ellos. Tercero, su sistema puede estar configurado para que los archivos ejecutables de ciertos comandos «restringidos» se mantengan en directorios que no están listados en `PATH`. Por ejemplo, puede haber un directorio */usr/games* en el que haya ejecutables que estén prohibidos durante las horas normales de trabajo.

[12] A menos que sea un comando incorporado (como *cd* y *print*), en cuyo caso el código es simplemente parte del archivo ejecutable para todo el shell.

Por lo tanto, es posible que desee añadir directorios al `PATH` por defecto que se obtiene al iniciar sesión. Digamos que ha creado un directorio *bin* bajo su directorio de inicio de sesión para sus propios scripts y programas de shell. Para añadir este directorio a su `PATH` para que esté allí cada vez que inicie sesión, ponga esta línea en su *.profile*:

```
PATH="$PATH:$HOME/bin"
```

Esto establece `PATH` a lo que era antes, seguido inmediatamente por dos puntos y `$HOME/bin` (su directorio *bin* personal). Este es un uso bastante típico. (El uso de `$HOME` permite al administrador del sistema mover tu directorio personal de lugar, sin que tengas que arreglar tu archivo *.profile*).

Hay un detalle adicional importante que hay que entender sobre cómo funciona el `PATH`. Tiene que ver con los elementos vacíos (o «nulos») en el `PATH`. Un elemento nulo puede ocurrir de tres maneras: colocando dos puntos al principio de la `PATH`, colocando dos puntos al final de la `PATH`, o colocando dos dos puntos adyacentes en medio de la `PATH`. El shell trata un elemento nulo en `PATH` como sinónimo de "", el directorio actual, y busca en cualquier directorio en el que se encuentre en ese punto de la búsqueda de ruta.

```
PATH=:$HOME/bin:/usr/bin:/usr/local/bin   # Buscar primero en el directorio actual
PATH=$HOME/bin:/usr/bin:/usr/local/bin:   # Ultima búsqueda en el directorio actual
PATH=$HOME/bin::/usr/bin:/usr/local/bin   # Segunda busqueda en el directorio actual
```

Por último, si necesita saber de qué directorio proviene un comando, no es necesario que busque en los directorios de su `PATH` hasta encontrarlo. El comando incorporado al shell *whence* imprime la ruta completa del comando que le das como argumento, o sólo el nombre del comando si es un comando incorporado en sí mismo (como *cd*), un alias o una función (como veremos en el Capítulo 4).

Consideraciones de Seguridad de PATH

La configuración de la variable `PATH` puede tener importantes implicaciones para la seguridad.

En primer lugar, tener el directorio actual en su ruta es un verdadero agujero de seguridad, especialmente para los administradores del sistema, y la cuenta `root` nunca debe tener un elemento nulo (o punto explícito) en su ruta de búsqueda. ¿Por qué? Piensa en alguien que crea un script de shell llamado, por ejemplo, *ls*, lo hace ejecutable y lo coloca en un directorio al que `root` podría acceder mediante *cd*, como */tmp*:

```
rm -f /tmp/ls   # Ocultar las pruebas
/bin/ls "$@"    # Ejecutar ls real
```

```
nasty stuff here  # Ejecutar silenciosamente otras cosas como root
```

Si `root` tiene el directorio actual primero en `PATH`, entonces `cd /tmp; ls` hace lo que el malhechor quiera, y `root` no se entera. (Esto se conoce en el mundo de la seguridad como un «troyano».) Aunque es menos grave para los usuarios normales, hay muchos expertos que todavía aconsejan no tener el directorio actual en `PATH`.

En segundo lugar, la forma más segura de añadir su papelera personal a `PATH` es al final. Cuando se introduce un comando, el shell busca en los directorios en el orden en que aparecen en PATH hasta que encuentra un archivo ejecutable. Por lo tanto, si tiene un script de shell o un programa cuyo nombre es el mismo que el de un comando existente, el shell utilizará el comando existente, a menos que escriba la ruta completa del comando para desambiguar. Por ejemplo, si ha creado su propia versión del comando *more* en `$HOME/bin` y su `PATH` tiene `$HOME/bin` al final, para obtener su versión deberá escribir `$HOME/bin/more` (o simplemente `~/bin/more`).

La forma más temeraria de reajustar tu ruta es decirle al shell que busque primero en tu directorio poniéndolo antes de los otros directorios en tu `PATH`:

```
PATH="$HOME/bin:$PATH"
```

Esto es menos seguro porque estás confiando en que tu propia versión del comando *more* funcione correctamente. Pero también es arriesgado, ya que podría permitir caballos de Troya (similar al ejemplo de *ls* que acabamos de ver). Si tu directorio *bin* es escribible por otros en tu sistema, pueden instalar un programa que haga algo desagradable. El uso apropiado de `PATH` es sólo uno de los muchos aspectos de la seguridad del sistema. Vea el Capítulo 10 para más detalles. En resumen, recomendamos dejar el directorio actual fuera de su `PATH` (tanto implícita como explícitamente), añadir su directorio *bin* personal al final de `PATH`, y asegurarse de que sólo usted puede crear, eliminar o cambiar archivos en su directorio *bin* personal.

PATH y Alias Rastreados

Vale la pena señalar que una búsqueda a través de los directorios en su `PATH` puede tomar tiempo. No morirá exactamente si aguanta la respiración durante el tiempo que la mayoría de los ordenadores tardan en buscar en su `PATH`, pero el gran número de operaciones de E/S de disco implicadas en algunas búsquedas en el `PATH` pueden llevar más tiempo que el que tarda en ejecutarse el comando invocado.

El shell de Korn proporciona una forma de eludir las búsquedas `PATH`, llamada alias rastreado. En primer lugar, observe que si especifica un comando indicando su ruta completa, el intérprete de órdenes ni siquiera utilizará su `PATH`, sino que irá directamente al archivo ejecutable. Los alias rastreados hacen esto automáticamente. La primera vez que invocas un comando, el shell busca el ejecutable de la forma normal (a través de `PATH`). A continuación, crea un alias para la ruta completa, de modo que la próxima vez que invoque el comando, el intérprete de comandos utilice la ruta completa y no se preocupe por el `PATH`. Si alguna vez cambias tu `PATH`, el shell marca los alias rastreados como «indefinidos», de modo que vuelve a buscar las rutas completas cuando invoques los comandos correspondientes.

De hecho, puedes añadir alias rastreados con el único propósito de evitar la búsqueda en el `PATH` de comandos que utilizas con especial frecuencia. Sólo tienes que poner un «alias trivial» de la forma `alias -t command` en tu archivo *.profile* o de entorno; el shell sustituye el nombre de ruta completo por sí mismo.

Por ejemplo, la primera vez que invocas *emacs*, el shell hace una búsqueda `PATH`. Al encontrar la ubicación de *emacs* (digamos */usr/local/bin/emacs*), el shell crea un alias rastreado:

```
alias -t emacs=/usr/local/bin/emacs   # Alias de seguimiento automatico
```

La próxima vez que ejecute *emacs*, el intérprete de comandos expande el alias *emacs* en la ruta completa /usr/local/bin/emacs, y ejecuta el programa directamente, sin molestarse con una búsqueda PATH.

También puede definir alias rastreados individuales usted mismo, con la opción *-t* del comando alias, y puede listar todos esos alias rastreados escribiendo `alias -t` por sí mismo. (Por compatibilidad con el shell de Bourne del Sistema V, *ksh* predefine el alias `hash='alias -t --'`; el comando hash en ese shell muestra la tabla interna de comandos encontrados. El mecanismo de alias rastreado del shell de Korn es más flexible).

Aunque la documentación del shell y la opción *trackall* indican que puede activar y desactivar el rastreo de alias, el comportamiento real del shell es diferente: el rastreo de alias siempre está activado. *alias -t* lista todos los alias rastreados creados automáticamente. Sin embargo, *alias -p* no imprime los alias rastreados. Esto se debe a que, conceptualmente, los alias rastreados son sólo una mejora del rendimiento; en realidad no están relacionados con los alias que se definen para la personalización.

Ruta de Búsqueda de Directorios

`CDPATH` es una variable cuyo valor, como el de `PATH`, es una lista de directorios separados por dos puntos. Su propósito es aumentar la funcionalidad del comando incorporado *cd*.

Por defecto, `CDPATH` no está definido (es decir, es nulo), y cuando se escribe *cd dirname*, el shell busca en el directorio actual un subdirectorio llamado *dirname*. De forma similar a `PATH`, esta búsqueda se desactiva cuando *dirname* comienza con una barra. Si establece `CDPATH`, le da al shell una lista de lugares donde buscar *dirname*; la lista puede o no incluir el directorio actual.

He aquí un ejemplo. Considere el alias para el comando *cd* largo de antes en este capítulo:

```
alias cdcm="cd work/projects/devtools/windows/confman"
```

Ahora supongamos que hay unos cuantos directorios bajo este directorio a los que necesitas ir a menudo; se llaman *src*, *bin* y *doc*. Usted define su `CDPATH` así:

```
CDPATH=:~/work/projects/devtools/windows/confman
```

En otras palabras, defina su `CDPATH` como la cadena vacía (es decir, el directorio actual, dondequiera que se encuentre) seguida de *~/work/projects/devtools/windows/confman*. Con esta configuración, si escribes `cd doc`, entonces el shell busca en el directorio actual un (sub)directorio llamado *doc*. Asumiendo que no encuentra ninguno, busca en el directorio */work/projects/devtools/windows/confman*. El shell encuentra el directorio doc allí, así que vas directamente a él.

Esto funciona para cualquier ruta relativa. Por ejemplo, si tienes un directorio *src/whizprog* en tu directorio personal, y tu `CDPATH` es `:$HOME` (el directorio actual y tu directorio personal), escribiendo cd *src/whizprog* te lleva a *$HOME/src/whizprog* desde cualquier parte del sistema.

Esta característica le da otra forma de ahorrar tecleando cuando necesita *cd* a menudo a directorios que están enterrados profundamente en su jerarquía de archivos. Usted puede encontrarse yendo a un grupo específico de directorios a menudo mientras trabaja en un proyecto en particular, y luego cambiando a otro grupo de directorios cuando cambia a otro proyecto. Esto implica que la característica `CDPATH` sólo es útil si la actualiza cada vez que cambien sus hábitos de trabajo; si no lo hace, puede encontrarse ocasionalmente donde no quiere estar.

Variables Misceláneas

Hemos cubierto las variables del shell que son importantes desde el punto de vista de la personalización. También hay varias que sirven como indicadores de estado y para otros propósitos diversos. Sus significados son relativamente sencillos; los más básicos se resumen en la Tabla 3.4.

Las primeras dos variables son establecidas por el programa de inicio de sesión, antes de que se inicie el shell. El shell establece el valor de las dos siguientes cada vez que usted cambia de directorio. El valor de la última variable cambia dinámicamente, a medida que transcurre el tiempo. Aunque también puede establecer los valores de cualquiera de estas variables, como cualquier otra, es difícil imaginar una situación en la que desee hacerlo.

Tabla 3.4: Variables de estado

Variable	Significado
`HOME`	Nombre de tu directorio de inicio (login). Este es el argumento por defecto para el comando `cd`.
`SHELL`	Nombre de ruta del shell que los programas deben utilizar para ejecutar comandos.
`PWD`	Directorio actual.
`OLDPWD`	Directorio anterior al último comando `cd`.
`SECONDS`	Número de segundos transcurridos desde que se invocó al shell.

3.5. Personalización y Subprocesos

Algunas de las variables discutidas anteriormente son utilizadas por los comandos que puede ejecutar, a diferencia del propio shell, para que puedan determinar ciertos aspectos de su entorno. La mayoría, sin embargo, ni siquiera se conocen fuera del caparazón.

Esta dicotomía plantea una pregunta importante: ¿qué «cosas» del caparazón se conocen fuera del caparazón y cuáles son solo internas? Esta pregunta está en el centro de muchos malentendidos sobre el shell y la programación del shell. Antes de responder, lo preguntaremos nuevamente de una manera más precisa: ¿qué «cosas» de shell son conocidas por los subprocesos? Recuerde que cada vez que ingresa un comando, le está diciendo al shell que ejecute ese comando en un subproceso; además, algunos programas complejos pueden iniciar sus propios subprocesos.

La respuesta es bastante simple. Los subprocesos heredan solo variables de entorno. Están disponibles automáticamente, sin que el subproceso tenga que realizar ninguna acción explícita. Todas las demás «cosas» (opciones de shell, alias y funciones) deben estar ex-

plícitamente disponibles. El archivo de entorno es cómo se hace esto. Además, solo los shells interactivos procesan el archivo de entorno. Las siguientes dos secciones describen las variables de entorno y el archivo de entorno, respectivamente.

3.5.1. Variables de Entorno

Por defecto, sólo una clase de cosas es conocida por todas las clases de subprocesos: una clase especial de variables del shell llamadas variables de entorno. Algunas de las variables incorporadas que hemos visto son en realidad variables de entorno: `HISTFILE`, `HOME`, `LOGNAME`, `PATH`, `PWD`, `OLDPWD`, `SHELL` y `TERM`.

Debería estar claro por qué estas y otras variables necesitan ser conocidas por los subprocesos. Ya hemos visto el ejemplo más obvio: los editores de texto como vi y Emacs necesitan saber qué tipo de terminal estás usando; `TERM` es su forma de determinarlo. Como otro ejemplo, la mayoría de los programas de correo de Unix le permiten editar un mensaje con su editor de texto favorito. ¿Cómo sabe *mail* qué editor usar? El valor de `EDITOR` (o a veces `VISUAL`).

Cualquier variable puede convertirse en una variable de entorno, y se pueden crear nuevas variables que sean variables de entorno. Las variables de entorno se crean con el comando:

```
export varnames
```

(*varnames* puede ser una lista de nombres de variables separados por espacios en blanco.) Si los nombres en *varnames* ya existen, entonces esas variables se convierten en variables de entorno. Si no existen, el shell crea nuevas variables que son variables de entorno.

Con *ksh*, puede asignar un valor y exportar la variable en un solo paso:

```
export TMPDIR=/var/tmp
```

También puedes definir variables para que estén sólo en el entorno de un subproceso (comando) en particular, precediendo al comando con la asignación de la variable, de la siguiente manera:

```
varname=value command
```

Puedes poner tantas asignaciones antes del comando como quieras.[13] Por ejemplo, supongamos que está utilizando el editor Emacs. Está teniendo problemas para que funcione con

[13]Existe una oscura opción, *keyword*, que (si está activada) te permite poner este tipo de definición de variable de entorno en cualquier parte de la línea de comandos, no sólo al principio.

su terminal, así que está experimentando con diferentes valores de *TERM*. Usted puede hacer esto más fácilmente mediante la introducción de comandos que se parecen:

```
TERM=trythisone emacs filename
```

emacs tiene *trythisone* definido como su valor de `TERM`, sin embargo la variable de entorno en su shell mantiene cualquier valor (si lo hubiera) que tuviera antes. Esta sintaxis no es muy usada, así que no la veremos muy a menudo en el resto de este libro.

No obstante, las variables de entorno son importantes. La mayoría de los archivos `.profile` incluyen definiciones de variables de entorno; el *.profile* de ejemplo anterior en este capítulo contenía dos de estas definiciones:

```
EDITOR=/usr/local/bin/emacs
SHELL=/bin/ksh
export EDITOR SHELL
```

Por alguna razón, el shell de Korn no hace de `EDITOR` una variable de entorno por defecto. Esto significa, entre otras cosas, que *mail* no sabrá qué editor usar cuando quieras editar un mensaje.[14] Por lo tanto, tendría que exportarlo usted mismo utilizando el comando *export* en su *.profile*.

La segunda línea del código anterior está pensada para sistemas que no tienen el shell de Korn instalado como shell por defecto, es decir, como */bin/sh*. Algunos programas ejecutan shells como subprocesos dentro de sí mismos (por ejemplo, muchos programas de correo y el modo shell del editor Emacs); por convención, utilizan la variable `SHELL` para determinar qué shell utilizar.

Puede averiguar qué variables son de entorno y cuáles son sus valores escribiendo *export* sin argumentos.

3.5.2. El Archivo de Entorno

Aunque las variables de entorno siempre son conocidas por los subprocesos, hay que indicar explícitamente al shell qué otras variables, opciones, alias, etc., deben comunicarse a los subprocesos. La forma de hacerlo es poner todas esas definiciones en un archivo especial llamado *archivo de entorno* en lugar de su *.profile*.

Puede llamar al archivo de entorno como desee, siempre que establezca la variable de entorno `ENV` con el nombre del archivo. La forma habitual de hacerlo es la siguiente:

[14] En realidad, por defecto será el editor de líneas *ed*. No quieres eso, ¿verdad?

1. Decide qué definiciones de tu *.profile* quieres propagar a los subprocesos. Elimínelas de *.profile* y colóquelas en un archivo que designe como su archivo de entorno.

2. Pon una línea en tu *.profile* que le diga al shell dónde está tu archivo de entorno:

   ```
   ENV=envfilename
   export ENV
   ```

   Es importante que el valor de `ENV` sea exportado, para que los subprocesos del shell puedan encontrarlo.

3. Para que los cambios surtan efecto inmediatamente, cierre la sesión y vuelva a iniciarla.[15] (No puede utilizar simplemente `. ~/.profile`; el shell no vuelve a ejecutar el archivo `$ENV` cuando cambia el valor de `ENV`).

La idea del fichero de entorno viene del fichero *.cshrc* del shell C; por lo tanto, muchos usuarios del shell de Korn que vinieron del mundo del shell C llaman a sus ficheros de entorno *.kshrc*. (El sufijo *rc* para los ficheros de inicialización es prácticamente universal en todo el mundo Unix. Significa «ejecutar comandos» y entró en el léxico de Unix a través del Sistema Compatible de Tiempo Compartido (CTSS) del MIT).

Como regla general, debería poner tan pocas definiciones como sea posible en *.profile* y tantas como sea posible en su fichero de entorno. Debido a que las definiciones agregan a un entorno en lugar de quitarlo, hay pocas posibilidades de que causen que algo en un subproceso no funcione correctamente. (Una excepción podrían ser los choques de nombres si te pasas con los alias).

Las únicas cosas que realmente necesitan estar en *.profile* son los comandos que no son definiciones sino que realmente se ejecutan o producen salida cuando te conectas. Las definiciones de opciones y alias deberían ir en el fichero de entorno. De hecho, hay muchos usuarios del shell de Korn que tienen archivos *.profile* diminutos, por ejemplo:

```
stty stop ^S intr ^C erase ^?
date
from
export ENV=~/.kshrc
```

(El comando *from*, en algunas versiones de Unix, comprueba si tiene correo e imprime una lista de las cabeceras de los mensajes en caso afirmativo). Aunque este es un pequeño *.profile*, el fichero de entorno de este usuario podría ser enorme.

[15] Esto asume que el shell de Korn está definido como su shell de inicio de sesión. Si no es así, debe hacer que el administrador del sistema lo instale como shell de inicio de sesión.

Hay una diferencia importante entre *ksh88* y *ksh93*. En *ksh88*, el fichero de entorno se ejecuta siempre. En *ksh93*, sólo los shells interactivos (aquellos que no leen desde un script, sino desde un terminal) ejecutan el fichero de entorno. Por lo tanto, es mejor que el archivo de entorno contenga sólo comandos que sean útiles para el uso interactivo, como la configuración de alias y opciones.

Otra diferencia entre las dos versiones del shell es que *ksh88* sólo hace sustitución de variables en el valor de *ENV*, mientras que *ksh93* hace sustitución de variables, comandos y aritmética en su valor. (La sustitución de comandos se describe en el Capítulo 4. La sustitución aritmética se describe en el Capítulo 5.) La sustitución aritmética se describe en el Capítulo 6).

3.6. Sugerencias de Personalización

No dudes en probar cualquiera de las técnicas presentadas en este capítulo. La mejor estrategia es probar algo escribiéndolo en el intérprete de comandos durante su sesión de inicio de sesión; si decide que quiere convertirlo en una parte permanente de su entorno, añádalo a su *.profile*.

Una forma agradable e indolora de añadir a su *.profile* sin entrar en un editor de texto hace uso del comando *print* y uno de los modos de edición del shell de Korn. Si escribe un comando de personalización y más tarde decide añadirlo a su *.profile*, puede recuperarlo mediante CTRL-P o CTRL-R (en modo *emacs*) o j, -, o / (modo vi). Digamos que la línea es:

```
PS1="($LOGNAME !)->"
```

Después de recuperarla, edítala para que esté precedida por un comando de *print*, rodeada por comillas simples, y seguida por una redirección de E/S que (como verás en el Capítulo 7) anexa la salida a *~/.profile*:

```
print 'PS1="($LOGNAME !)->"' >> ~/.profile
```

Recuerde que las comillas simples son importantes porque evitan que el shell intente interpretar cosas como signos de dólar, comillas dobles y signos de exclamación.

También deberías sentirte libre de husmear en los archivos *.profile* de otras personas para obtener ideas de personalización. Una forma rápida de examinar el *.profile* de todo el mundo es la siguiente: supongamos que todos los directorios de inicio de sesión están en */home*. Entonces puedes escribir:

```
cat /home/*/.profile > ~/other\.profiles
```

y examinar los archivos *.profile* de otras personas con un editor de texto a tu antojo (suponiendo que tengas permiso de lectura sobre ellos). Si otros usuarios tienen archivos de entorno, el archivo que acabas de crear mostrará cuáles son y también podrás examinarlos.

Por último, asegúrate de que nadie más que tú tiene permisos de escritura en tus archivos *.profile* y de entorno.

CAPÍTULO 4

PROGRAMACIÓN BÁSICA DE SHELL

Si te has familiarizado con las técnicas de personalización que presentamos en el capítulo anterior, probablemente te habrás encontrado con varias modificaciones en tu entorno que quieres hacer pero no puedes - todavía. La programación Shell las hace posibles.

El shell de Korn tiene algunas de las capacidades de programación más avanzadas de cualquier intérprete de comandos de su tipo. Aunque su sintaxis no es ni de lejos tan elegante o consistente como la de la mayoría de los lenguajes de programación convencionales, su potencia y flexibilidad son comparables. De hecho, el shell de Korn puede utilizarse como un entorno completo para escribir prototipos de software.

Algunos aspectos de la programación en el shell de Korn son realmente extensiones de las técnicas de personalización que ya hemos visto, mientras que otros se asemejan a las características de los lenguajes de programación tradicionales. Hemos estructurado este capítulo de forma que si no eres programador, puedes leer este capítulo y hacer bastante más de lo que podrías hacer con la información del capítulo anterior. La experiencia con un lenguaje de programación convencional como Pascal o C es útil (aunque no estrictamente necesaria) para los capítulos siguientes. A lo largo del resto del libro, nos encontraremos ocasionalmente con problemas de programación, llamados «tareas», cuyas soluciones hacen uso de los conceptos que cubrimos.

4.1. Scripts y Funciones del Shell

Un *script*, o archivo que contiene comandos del shell, es un programa del shell. Sus archivos *.profile* y de entorno, discutidos en el Capítulo 3, son scripts de shell.

Puedes crear un script utilizando el editor de texto que prefieras. Una vez que hayas creado uno, hay varias maneras de ejecutarlo. Una, que ya hemos cubierto, es escribir .

`scriptname` (es decir, el comando es un punto). Esto hace que los comandos del script sean leídos y ejecutados como si los hubieras tecleado.

Otras dos formas son escribir `ksh script` o `ksh < script`. Estos invocan explícitamente el shell de Korn en el script, requiriendo que usted (y sus usuarios) sean conscientes de que son scripts.

La última forma de ejecutar un script es simplemente escribir su nombre y pulsar ENTER, como si estuviera invocando un comando integrado. Esta, por supuesto, es la forma más conveniente. Este método hace que el script se parezca a cualquier otro comando Unix, y de hecho varios comandos «normales» se implementan como scripts de shell (es decir, no como programas escritos originalmente en C o algún otro lenguaje), incluyendo *spell*, *man* en algunos sistemas, y varios comandos para administradores de sistemas. La consiguiente falta de distinción entre «archivos de comandos de usuario» y «comandos incorporados» es uno de los factores que explican la extensibilidad de Unix y, por tanto, su favoritismo entre los programadores.

Puede ejecutar un script tecleando su nombre sólo si . (el directorio actual) forma parte de su ruta de búsqueda de comandos, es decir, si está incluido en su variable PATH (como se explica en el Capítulo 3). Si . no está en su ruta, debe teclear `./ nombredelscript`, que es realmente lo mismo que teclear la ruta relativa del script (ver Capítulo 1).

Antes de que pueda invocar el script de shell por su nombre, también debe darle permiso de «ejecución». Si está familiarizado con el sistema de archivos Unix, sabrá que los archivos tienen tres tipos de permisos (lectura, escritura y ejecución) y que esos permisos se aplican a tres categorías de usuarios (el propietario del archivo, un grupo de usuarios y todos los demás). Normalmente, cuando creas un archivo con un editor de texto, el archivo se configura con permiso de lectura y escritura para ti y permiso de sólo lectura para todos los demás[1].

Por lo tanto, debe dar a su script permiso de ejecución explícitamente, utilizando el comando *chmod(1)*. La forma más sencilla de hacerlo es así:

```
chmod +x scriptname
```

Su editor de texto conserva este permiso si realiza cambios posteriores en el script. Si no añade permiso de ejecución al script, e intenta invocarlo, el shell imprime el mensaje:

```
ksh: scriptname: cannot execute [Permission denied]
```

[1]En realidad, esto depende de la configuración de tu *umask*, una función avanzada que se describe en el Capítulo 10.

Pero hay una diferencia más importante entre las dos formas de ejecutar scripts de shell. Mientras que el método *punto* «dot» hace que los comandos del script se ejecuten como si fueran parte de tu sesión de inicio de sesión, el método «sólo el nombre» hace que el shell haga una serie de cosas. En primer lugar, ejecuta otra copia del shell como subproceso. El subproceso del intérprete de comandos toma los comandos del script, los ejecuta y termina, devolviendo el control al intérprete de comandos principal.

La Figura 4.1 muestra cómo el shell ejecuta scripts. Suponga que tiene un script de shell simple llamado *fred* que contiene los comandos *bob* y *dave*. En la Figura 4.1.a, escribir `. fred` hace que los dos comandos se ejecuten en el mismo shell, como si los hubieras escrito a mano. La Figura 4-1.b muestra lo que sucede cuando escribes sólo `fred:` los comandos se ejecutan en el subproceso del shell mientras el shell padre espera a que el subproceso termine.

Puede resultarle interesante comparar esta situación con la de la Figura 4.1.c que muestra lo que ocurre cuando escribe `fred &`. Como recordará del Capítulo 1, el & hace que el comando se ejecute en segundo plano, que en realidad es sólo otro término para «subproceso». Resulta que la única diferencia significativa entre la Figura 4.1.c y la Figura 4.1.b es que usted tiene el control de su terminal o estación de trabajo mientras se ejecuta el comando - no necesita esperar a que termine para poder introducir más comandos.

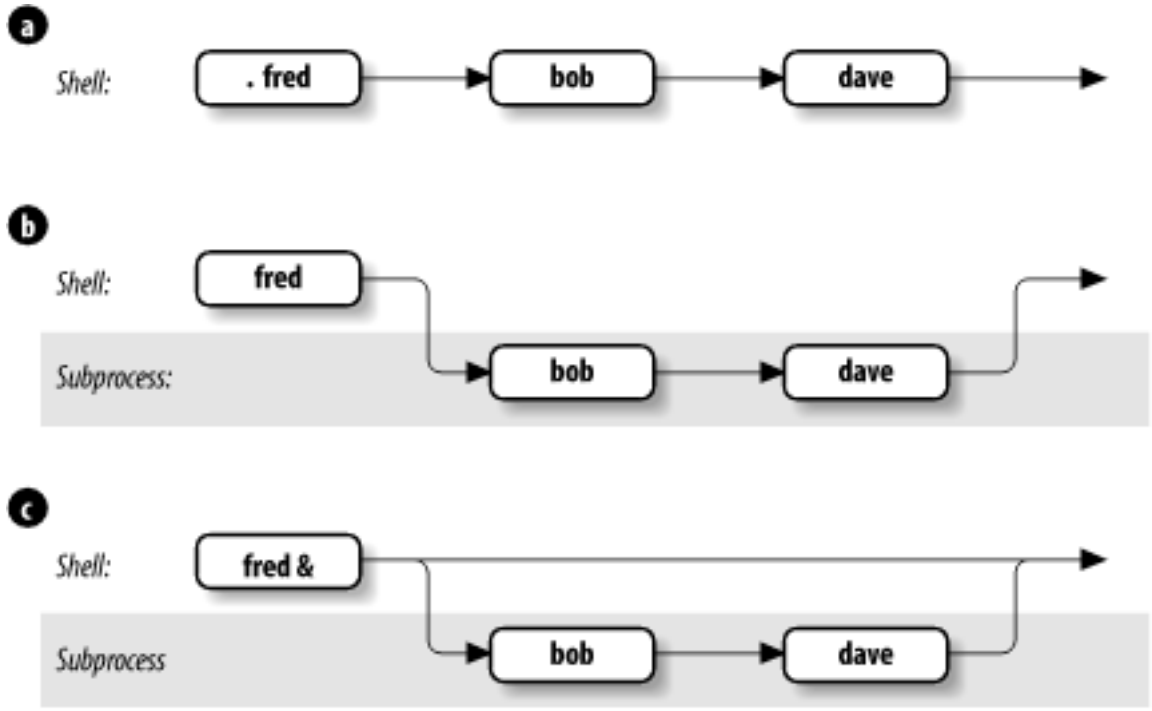

Figura 4.1: Formas de ejecutar un script de shell

El uso de subprocesos de shell tiene muchas ramificaciones. Una importante es que las varia-

bles de entorno exportadas que vimos en el último capítulo (por ejemplo, `TERM, LOGNAME, PWD`) son conocidas en los subprocesos del shell, mientras que otras variables del shell (como cualquiera que definas en tu *.profile* sin una sentencia *export*) no lo son.

Otras cuestiones relacionadas con los subprocesos del shell son demasiado complejas para entrar en ellas ahora; consulte el Capítulo 7 y el Capítulo 8 para obtener más detalles sobre la E/S de los subprocesos y las características de los procesos, respectivamente. Por ahora, sólo tenga en cuenta que un script normalmente se ejecuta en un subproceso del shell.

4.1.1. Funciones

La característica de función del shell de Korn es una versión ampliada de una facilidad similar en el shell de Bourne del Sistema V y algunos otros shells. Una función es una especie de script dentro de un script; se usa para definir algún código del shell por su nombre y almacenarlo en la memoria del shell, para ser invocado y ejecutado más tarde.

Las funciones mejoran significativamente la programabilidad del shell, por dos razones principales. En primer lugar, cuando invoca una función, ésta ya se encuentra en la memoria del shell (excepto en el caso de las funciones cargadas automáticamente; consulte la Sección 4.1.1.1, más adelante en este capítulo); por lo tanto, una función se ejecuta más rápidamente. Los ordenadores modernos tienen mucha memoria, así que no hay necesidad de preocuparse por la cantidad de espacio que ocupa una función típica. Por esta razón, la mayoría de la gente define tantas funciones como sea posible en lugar de tener muchos scripts a su alrededor.

La otra ventaja de las funciones es que son ideales para organizar largos scripts de shell en «trozos» modulares de código que son más fáciles de desarrollar y mantener. Si no eres programador, pregúntale a uno cómo sería la vida sin funciones (también llamadas *procedimientos* o *subrutinas* en otros lenguajes) y probablemente te echarán la bronca.

Para definir una función, puede utilizar una de estas dos formas:

```
function functname {          Semántica del shell de Korn
  Comandos de shell
}
```

O

```
function () {                 Semántica POSIX
  Comandos de shell
}
```

La primera forma proporciona acceso a toda la potencia y programabilidad del shell de Korn. La segunda es compatible con la sintaxis para funciones de shell introducida en el shell de Bourne de la versión 2 del sistema V. Esta forma obedece a la semántica del estándar POSIX, que es menos potente que las funciones completas estilo shell de Korn. (En este libro utilizaremos siempre la primera forma. Puede borrar una definición de función con el comando `unset -f nombrefuncion`.

Cuando define una función, le indica al shell que almacene su nombre y definición (es decir, los comandos del shell que contiene) en la memoria. Si desea ejecutar la función más tarde, sólo tiene que escribir su nombre seguido de los argumentos, como si se tratara de un script del shell.

Puede averiguar qué funciones están definidas en su sesión de inicio de sesión escribiendo `functions`[2]. (Observe la s al final del nombre del comando.) El shell imprimirá no sólo los nombres sino también las definiciones de todas las funciones, en orden alfabético por nombre de función. Dado que esto puede resultar en una salida larga, es posible que desee canalizar la salida a través de más o redirigirla a un archivo para su examen con un editor de texto.

Aparte de las ventajas, hay dos diferencias importantes entre las funciones y los scripts. En primer lugar, las funciones no se ejecutan en procesos separados, como hacen los scripts cuando los invocas por su nombre; la «semántica» de la ejecución de una función es más parecida a la de tu perfil *.profile* cuando te conectas o a la de cualquier script cuando se invoca con el comando «dot». En segundo lugar, si una función tiene el mismo nombre que un script o programa ejecutable, la función tiene preferencia.

Este es un buen momento para mostrar el orden de precedencia de las distintas fuentes de comandos. Cuando escribes un comando en el shell, éste busca en los siguientes lugares hasta que encuentra una coincidencia:

1. Palabras clave, como *function* y algunas otras (por ejemplo, *if* y *for*) que veremos en el Capítulo 5.

2. Alias (aunque no puede definir un alias cuyo nombre sea una palabra clave del shell, puede definir un alias que se expanda a una palabra clave, por ejemplo, `alias aslongas=while`; consulte el Capítulo 7 para obtener más detalles)

3. Complementos especiales, como *break* y *continue* (la lista completa es `. (punto), :,`

[2]Se trata en realidad de un alias de *typeset -f*; véase el capítulo 6.

`alias, break, continue, eval, exec, exit, export, login, newgrp, readonly, return, set, shift, trap, typeset, unalias, y unset`)

4. Funciones
5. Funciones incorporadas no especiales, como *cd* y *whence*
6. Scripts y programas ejecutables, que el shell busca en los directorios enumerados en la variable de entorno PATH.

Examinaremos este proceso con más detalle en la sección sobre procesamiento desde la línea de comandos del Capítulo 7.

Si necesita saber la fuente exacta de un comando, hay una opción para el comando incorporado *whence* que vimos en el Capítulo 3. *whence* por sí mismo imprimirá la ruta de un comando si el comando es un script o un programa ejecutable, pero sólo repetirá el nombre del comando si es cualquier otra cosa. Pero si escribe `whence -v nombredelcomando`, obtendrá información más completa, como por ejemplo:

```
$ whence -v cd
cd is a shell builtin
$ whence -v function
function is a keyword
$ whence -v man
man is a tracked alias for /usr/bin/man
$ whence -v ll
ll is an alias for 'ls -l'
```

Por compatibilidad con el shell de Bourne del Sistema V, el shell de Korn predefine el alias `type='whence -v'`. Esto definitivamente hace la transición al shell de Korn más fácil para los antiguos usuarios del shell de Bourne; *type* es similar a *whence*. El comando whence en realidad tiene varias opciones, descritas en la Tabla 4.1.

Tabla 4.1: Opciones del comando whence

Opción	**Significado**
-a	Imprime todas las interpretaciones del nombre dado.
-f	Omitir funciones en la búsqueda del nombre.
-p	Busca en $PATH, incluso si el nombre es un *built-in* o una función.
-v	Imprime una descripción más detallada del nombre.

En el resto de este libro nos referiremos principalmente a los scripts, pero a menos que indiquemos lo contrario, debes asumir que todo lo que digamos se aplica igualmente a las funciones.

Carga Automática de Funciones

A primera vista, parecería que el mejor lugar para poner sus propias definiciones de funciones es en su archivo *.profile* o de entorno. Esto es genial para el uso interactivo, ya que su shell de inicio de sesión lee `~/.profile`, y otros shells interactivos leen el fichero de entorno. Sin embargo, cualquier script de shell que escriba no lee ninguno de los dos archivos. Además, a medida que tu colección de funciones crece, también lo hacen tus archivos de inicialización, haciendo que sea difícil trabajar con ellos.

ksh93 soluciona estos dos problemas integrando la búsqueda de funciones con la búsqueda de comandos. Así es como funciona:

1. Cree un directorio para guardar las definiciones de sus funciones. Este puede ser su directorio bin privado, o puede que desee tener un directorio separado, como `~/funcs`. Por el bien de la discusión, supongamos esto último.

2. En su archivo *.profile*, añada este directorio a las variables PATH y FPATH:

```
PATH=$PATH:~/funcs
FPATH=~/funcs
export PATH FPATH
```

3. En `~/funcs`, coloque la definición de cada una de sus funciones en un archivo separado. El archivo de cada función debe tener el mismo nombre que la función:

```
$ mkdir ~/funcs
$ cd ~/funcs
$ cat > whoson
  # whoson --- create a sorted list of logged-on users
        function whoson {
          who | awk '{ print $1 }' | sort -u
        }
        ^D
```

Ahora, la primera vez que escriba `whoson`, el shell buscará un comando llamado whoson usando el orden de búsqueda descrito anteriormente. No se encontrará como un *built-in* especial, como una función o como un *built-in* normal. El intérprete de comandos inicia entonces una búsqueda a lo largo de `$PATH`. Cuando finalmente encuentra *~/funcs/whoson*, el shell se da cuenta de que *~/funcs* también está en `$FPATH`. (”¡Ajá!” dice el shell.) Cuando este es el caso, el shell espera encontrar la definición de la función llamada *whoson* dentro del archivo. Lee y ejecuta todo el contenido del archivo y sólo entonces ejecuta la función *whoson*, con cualquier argumento suministrado. (Si el archivo encontrado tanto en `$PATH` como en `$FPATH` no define realmente la función, obtendrá un mensaje de error «no

encontrado»).

La próxima vez que escriba *whoson*, la función ya estará definida, por lo que el shell la encontrará inmediatamente, sin necesidad de buscar la ruta.

Tenga en cuenta que los directorios listados en FPATH pero no en PATH no serán buscados por funciones, y que a partir de *ksh93l*, el directorio actual debe ser listado en FPATH mediante un punto explícito; dos puntos iniciales o finales no hacen que se busque en el directorio actual.

Por último, a partir de *ksh93m*, cada directorio nombrado en PATH puede contener un archivo llamado *.paths*. Este archivo puede contener comentarios y líneas en blanco, y asignaciones de variables especializadas. La primera asignación permitida es a FPATH, donde el valor debe nombrar un directorio existente. Si ese directorio contiene un archivo cuyo nombre coincide con la función que se está buscando, ese archivo se lee y se ejecuta como si fuera a través del comando . (dot), y luego se ejecuta la función.

Además, se puede asignar otra variable de entorno. Su uso previsto es especificar una ruta relativa o absoluta para un directorio de bibliotecas que contenga las bibliotecas compartidas para los ejecutables en el directorio bin actual. En muchos sistemas Unix, esta variable es `LD_LIBRARY_PATH`, pero algunos sistemas tienen una variable diferente - consulte su documentación local. El valor dado se añade al valor existente de la variable cuando se ejecuta la orden. (Este mecanismo puede abrir brechas de seguridad. Los administradores de sistemas deben utilizarlo con precaución).

Por ejemplo, el grupo AT&T Advanced Software Tools que distribuye *ksh93* también tiene muchas otras herramientas, a menudo instaladas en un directorio *ast/bin* separado. Esta característica permite a los programas ast encontrar sus bibliotecas compartidas, sin que el usuario tenga que ajustar manualmente `LD_LIBRARY_PATH` en el archivo *.profile*[3]. Por ejemplo, si un comando se encuentra en */usr/local/ast/bin*, y el archivo *.paths* en ese directorio contiene la asignación `LD_LIBRARY_PATH=../lib`, el shell añade `/usr/local/ast/lib:` al valor de `LD_LIBRARY_PATH` antes de ejecutar el comando.

Los lectores familiarizados con *ksh88* notarán que esta parte del comportamiento del shell ha cambiado significativamente. Dado que *ksh88* siempre leía el fichero de entorno, tanto si el shell era interactiva como si no, lo más sencillo era limitarse a poner allí las definiciones

[3]Las versiones h a l+ de *ksh93* utilizaban un mecanismo similar pero más restringido, a través de un fichero llamado *.fpath*, e incorporaban la configuración de la variable de ruta de la biblioteca. Como esta característica no estaba muy extendida, se generalizó en un único archivo a partir de la versión m.

de las funciones. Sin embargo, esto podía dar lugar a un archivo grande y difícil de manejar. Para evitar esto, puedes crear archivos en uno o más directorios listados en `$FPATH`. Luego, en el archivo de entorno, se marcarían las funciones como de *carga automática*:

```
autoload whoson
...
```

Marcar una función con *autoload*[4] le dice al shell que este nombre es una función, y que encuentre la definición buscando en `$FPATH`. La ventaja de esto es que la función no se carga en la memoria del shell si no se necesita. La desventaja es que tienes que listar explícitamente todas tus funciones en tu fichero de entorno.

La integración de *ksh93* de la búsqueda en PATH y FPATH simplifica así la forma de añadir funciones del intérprete de órdenes a su «biblioteca» personal de funciones del intérprete de órdenes.

Funciones POSIX

Como se mencionó anteriormente, las funciones definidas usando la sintaxis POSIX obedecen a la semántica POSIX y no a la semántica del shell de Korn:

```
functname () {
  comandos de shell
}
```

La mejor manera de entender esto es pensar en una función POSIX como si fuera un punto script. Las acciones dentro del cuerpo de la función afectan a todo el estado del script actual. En contraste, las funciones del shell de Korn tienen mucho menos estado compartido con el shell padre, aunque no son idénticas a scripts totalmente separados.

Los detalles técnicos siguen; incluyen información que aún no hemos cubierto. Así que vuelva y relea esta sección después de haber aprendido sobre el comando *typeset* en el Capítulo 6 y sobre traps en el Capítulo 8.

- Las funciones POSIX comparten variables con el script padre. Las funciones del shell de Korn pueden tener sus propias variables locales.
- Las funciones POSIX comparten trampas con el script padre. Las funciones del shell de Korn pueden tener sus propias trampas locales.
- Las funciones POSIX no pueden ser recursivas (llamarse a sí mismas)[5]. Las funciones

[4] *autoload* es en realidad un alias de *typeset -fu*.
[5] Esta es una restricción impuesta por el shell de Korn, no por el estándar POSIX.

del shell de Korn sí pueden.

- Cuando se ejecuta una función POSIX, `$0` no se cambia por el nombre de la función.

Si utiliza el comando *dot* con el nombre de una función del shell de Korn, dicha función obedecerá la semántica POSIX, afectando a todo el estado (variables y traps) del shell padre:

```
$ function demo {                          # Define una función de Korn Shell
>   typeset myvar=3                        # Establece una variable local llamada myvar
>   print "demo: myvar is $myvar"
> }
$ myvar=4                                  # Establece una variable global llamada myvar
$ demo ; print "global: myvar is $myvar"   # Ejecuta la función
demo: myvar is 3
global: myvar is 4
$ . demo                                   # Ejecuta la función con semantica POSIX
demo: myvar is 3
$ print "global: myvar is $myvar"          # Observa los resultados
global: myvar is 3
```

4.2. Variables del Shell

Una parte importante de la funcionalidad de programación del shell de Korn está relacionada con las variables del shell. Ya hemos visto los conceptos básicos de las variables. Para recapitular brevemente: son lugares con nombre para almacenar datos, normalmente en forma de cadenas de caracteres, y sus valores pueden obtenerse precediendo sus nombres con signos de dólar ($). Ciertas variables, llamadas variables de entorno, se nombran convencionalmente en mayúsculas, y sus valores se dan a conocer (con la sentencia export) a los subprocesos.

Esta sección presenta los conceptos básicos de las variables del shell. La discusión de ciertas características avanzadas se retrasa hasta más adelante en el capítulo, después de cubrir las expresiones regulares.

Si es programador, ya sabe que casi todos los lenguajes de programación importantes utilizan variables de alguna manera; de hecho, una forma importante de caracterizar las diferencias entre lenguajes es comparar sus facilidades para las variables.

La principal diferencia entre el esquema de variables del shell de Korn y los de los lenguajes convencionales es que el esquema del shell de Korn pone mucho énfasis en las cadenas de caracteres. (Por tanto, tiene más en común con un lenguaje de propósito especial como

SNOBOL que con uno de propósito general como Pascal). Esto también es cierto para el shell de Bourne y el shell C, pero el shell de Korn va más allá de ellos al tener mecanismos adicionales para manejar enteros y números de coma flotante de doble precisión explícitamente, así como arrays simples.

4.2.1. Parámetros de Posición

Como ya hemos visto, puedes definir valores para variables con sentencias de la forma `varname = value`, por ejemplo:

```
$ fred=bob
$ print ''$fred>>
bob
```

Algunas variables de entorno son predefinidas por el shell cuando te conectas. Hay otras variables incorporadas que son vitales para la programación del shell. Ahora veremos algunas de ellas y dejaremos las otras para más adelante.

Las variables incorporadas especiales más importantes se denominan *parámetros posicionales*. Contienen los argumentos de la línea de comandos de los scripts cuando son invocados. Los parámetros posicionales tienen los nombres `1, 2, 3,` etc., lo que significa que sus valores se denotan por `$1, $2, $3,` etc. También hay un parámetro posicional `0`, cuyo valor es el nombre del script (es decir, el comando tecleado para invocarlo).

Dos variables especiales contienen todos los parámetros posicionales (excepto el parámetro posicional `0`): * y @. La diferencia entre ellas es sutil pero importante, y sólo es evidente cuando están entre comillas dobles.

`''$*''` es una sola cadena que consta de todos los parámetros posicionales, separados por el primer carácter de la variable IFS (separador interno de campos), que por defecto es un espacio, TAB y una nueva línea. Por otra parte, `''$@''` es igual a `''$1>> ''$2>> ... ''$ N ''`, donde *N* es el número de parámetros posicionales. Es decir, es igual a *N* cadenas separadas por comillas dobles, separadas por espacios. Dentro de un rato exploraremos las ramificaciones de esta diferencia.

La variable # contiene el número de parámetros posicionales (como una cadena de caracteres). Todas estas variables son de «sólo lectura», lo que significa que no puede asignarles nuevos valores dentro de los scripts. (Pueden cambiarse, pero no mediante asignación. Consulte la Sección 4.2.1.2, más adelante en este capítulo).

Por ejemplo, suponga que tiene el siguiente script de shell simple:

```
print "fred: $*''
print ''$0: $1 and $2>>
print ''$# arguments>>
```

Supongamos además que el script se llama *fred*. Entonces, si escribe `fred bob dave`, verá la siguiente salida:

```
b dave
fred: bob and dave
2 arguments
```

En este caso, `$3, $4,` etc., no están definidos, lo que significa que el shell los sustituye por la cadena vacía (o nula) (a menos que la opción *nounset* esté activada).

Parámetros de Posición en las Funciones

Las funciones de shell utilizan parámetros posicionales y variables especiales como * y # exactamente del mismo modo que los scripts de shell. Si quisieras definir *fred* como una función, podrías poner lo siguiente en tu archivo *.profile* o de entorno:

```
function fred {
  print "fred: $*"
  print ''$0: $1 and $2''
  print ''$# arguments''
}
```

Obtendrás el mismo resultado si escribes `fred bob dave`.

Normalmente, se definen varias funciones de shell dentro de un único script de shell. Por lo tanto, cada función necesita manejar sus propios argumentos, lo que a su vez significa que cada función necesita hacer un seguimiento de los parámetros posicionales por separado. Por supuesto, cada función tiene sus propias copias de estas variables (aunque las funciones no se ejecuten en su propio subproceso, como los scripts); decimos que tales variables son locales a la función.

Otras variables definidas dentro de las funciones no son locales; son globales, lo que significa que sus valores son conocidos a través de todo el script de shell[6]. Por ejemplo, suponga que tiene un script de shell llamado *ascript* que contiene esto:

```
function afunc {
  print in function $0: $1 $2
  var1="in function"
}
var1="outside of function"
```

[6]Sin embargo, en la sección sobre composición tipográfica del Capítulo 6 se explica cómo hacer que las variables sean locales a las funciones.

```
print var1: $var1
print $0: $1 $2
afunc funcarg1 funcarg2
print var1: $var1
print $0: $1 $2
```

Si invocas este script escribiendo `ascript arg1 arg2`, verás esta salida:

```
var1: outside of function
ascript: arg1 arg2
in function afunc: funcarg1 funcarg2
var1: in function
ascript: arg1 arg2
```

En otras palabras, la función *afunc* cambia el valor de la variable `ivar1` de «fuera de la función» a «en la función», y ese cambio se conoce fuera de la función, mientras que `$0, $1` y `$2` tienen valores diferentes en la función y en el script principal. La Figura 4.2 muestra esto gráficamente.

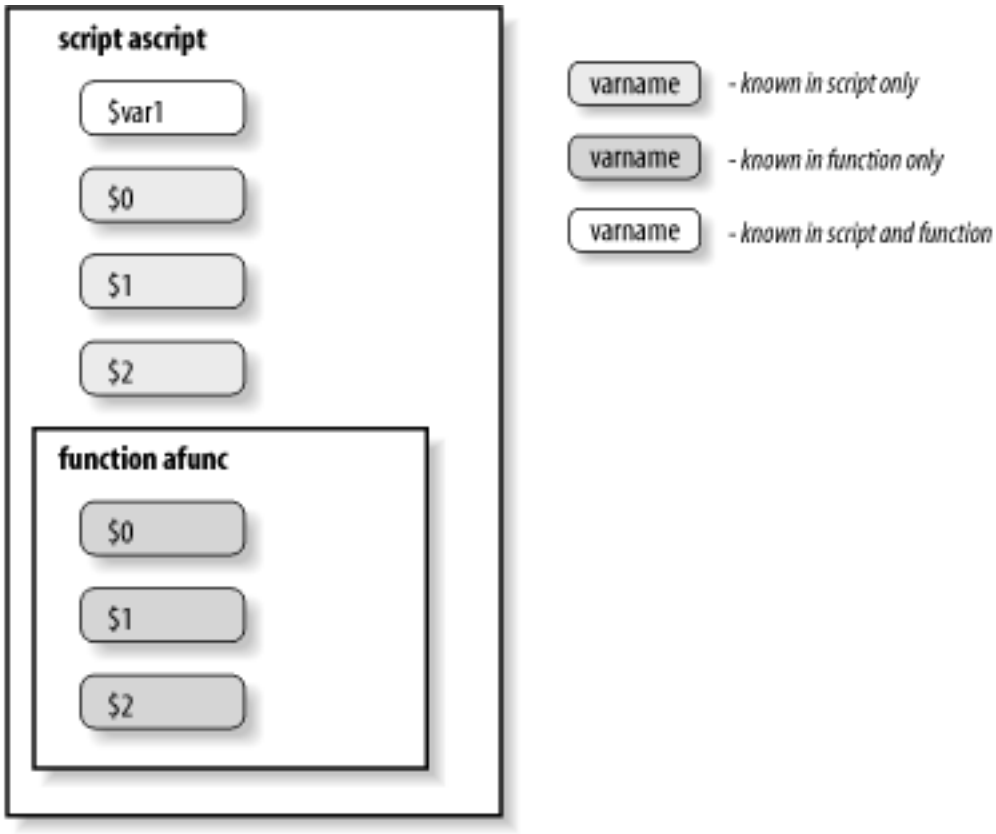

Figura 4.2: Las funciones tienen suspropios parámetros de posición

Es posible hacer otras variables locales a funciones usando el comando *typeset*, que veremos en el Capítulo 6. Ahora que tenemos estos antecedentes, echemos un vistazo más de cerca a "$@" y "$*". Estas variables son dos de las mayores idiosincrasias del shell, así que discutiremos algunas de las fuentes más comunes de confusión.

- ¿Por qué los elementos de `''$*''` están separados por el primer carácter de IFS en lugar de sólo por espacios? Para darle flexibilidad de salida. Como ejemplo simple, digamos que desea imprimir una lista de parámetros posicionales separados por comas. Este script lo haría:

```
FS=,
print "$*"
```

  Cambiar IFS en un script es bastante arriesgado, pero probablemente esté bien mientras nada más en el script dependa de ello. Si este script se llamara *arglist*, el comando `arglist bob dave ed` produciría la salida `bob,dave,ed`. El Capítulo 10 contiene otro ejemplo de cambio de IFS.

- ¿Por qué `''$@''` actúa como *N* cadenas separadas entre comillas dobles? Para permitirle utilizarlas de nuevo como valores separados. Por ejemplo, digamos que desea llamar a una función dentro de su script con la misma lista de parámetros posicionales, así:

```
function countargs {
  print "$# args."
}
```

  Asuma que su script es llamado con los mismos argumentos que *arglist* arriba. Entonces si contiene el comando `countargs ''$*''`, la función imprime `1 args`. Pero si el comando es `countargs ''$@''`, la función imprime `3 args`.

  Poder recuperar los argumentos tal y como entraron también es importante en caso de que necesite preservar cualquier espacio en blanco incrustado. Si tu script fue invocado con los argumentos «hi», «howdy», y «hello there», aquí están los diferentes resultados que podrías obtener:

```
$ countargs $*
4 args
$ countargs "$*"
1 args
$ countargs $@
4 args
$ countargs "$@"
3 args
```

  Dado que ”$@” siempre conserva exactamente los argumentos, lo utilizamos en casi todos los programas de ejemplo de este libro.

Modificación de los Parámetros de Posición

En ocasiones, resulta útil modificar los parámetros de posición. Ya hemos mencionado que no se pueden establecer directamente, utilizando una asignación como `1="primero"`. Sin embargo, el conjunto de comandos incorporado puede utilizarse para este propósito.

El comando *set* es quizás el comando más complicado y sobrecargado del shell. Toma un gran número de opciones, las cuales se discuten en el Capítulo 9. Lo que nos importa por el momento es que los argumentos adicionales no opcionales para *set* reemplazan a los parámetros posicionales. Supongamos que nuestro script fue invocado con los tres argumentos «bob», «fred», y «dave». Entonces `countargs "$@"` nos dice que tenemos tres argumentos. Al usar *set* para cambiar los parámetros posicionales, `$#` se actualiza también.

```
$ set one two three "four not five"      # Modificar los parametros de posicion
$ countargs "$@"                         # Verificar el cambio
4 args
```

El comando *set* también funciona dentro de una función shell. Los parámetros de posición de la función shell se modifican, pero no los del script de llamada:

```
$ function testme {
>     countargs "$@"                     # Mostrar el numero original de parametros
>     set a b c                          # Ahora cambiarlos
>     countargs "$@"                     # Imprimir el nuevo recuento

> }
$ testme 1 2 3 4 5 6                     # Ejecuta la funcion
6 args                                   # Recuento original
3 args                                   # Nuevo recuento
$ countargs "$@"                         # Sin cambios en los parametros del shell invocadora
4 args
```

4.2.2. Más Sobre Sintaxis de Variables

Antes de mostrar las muchas cosas que puedes hacer con las variables del shell, tenemos que hacer una confesión: la sintaxis de `$ varname` para tomar el valor de una variable no es del todo exacta. En realidad, es la forma simple de la sintaxis más general, que es `$varname`.

¿Por qué dos sintaxis? Por un lado, la sintaxis más general es necesaria si su código se refiere a más de nueve parámetros posicionales: debe usar `$10` para el décimo en lugar de `$10`. (Esto asegura la compatibilidad con el shell de Bourne, donde `$10` significa `$10`.) Aparte de eso, considere el ejemplo del Capítulo 3 de establecer su variable prompt primaria (PS1)

a su nombre de usuario:

```
PS1="($LOGNAME)-> "
```

Esto funciona porque el paréntesis derecho que sigue inmediatamente a `LOGNAME` no es un carácter válido para un nombre de variable, por lo que el shell no lo confunde con parte del nombre de la variable. Ahora suponga que, por alguna razón, quiere que su prompt sea su nombre de usuario seguido de un guión bajo. Si escribe:

```
PS1="$LOGNAME_ "
```

entonces el shell intenta utilizar `''LOGNAME_''` como nombre de la variable, es decir, tomar el valor de `$LOGNAME_`. Como no existe tal variable, el valor por defecto es *null* (la cadena vacía, ""), y PS1 se establece sólo con un espacio.

Por esta razón, la sintaxis completa para tomar el valor de una variable es `$ varname` . Así que si usamos:

```
PS1="${LOGNAME}_ "
```

obtendríamos el deseado `tunombre_`. Es seguro omitir las llaves ({}) si el nombre de la variable va seguido de un carácter que no sea una letra, un dígito o un guión bajo.

4.2.3. Añadir una Variable

Como se ha mencionado, las variables del shell de Korn tienden a estar orientadas a cadenas. Una operación muy común es añadir un nuevo valor a una variable existente. (Por ejemplo, reunir un conjunto de opciones en una sola cadena.) Desde tiempos inmemoriales, esto se hacía aprovechando la sustitución de variables dentro de comillas dobles:

```
myopts="$myopts $newopt"
```

Los valores de `myopts` y `newopt` se concatenan en una sola cadena, y el resultado se asigna de nuevo a `myopts`. A partir de *ksh93j*, el shell de Korn proporciona un mecanismo más eficiente e intuitivo para hacer esto:

```
myopts+=" $newopt"
```

Esto consigue lo mismo, pero es más eficiente, y además deja claro que el nuevo valor se está añadiendo a la cadena. (En C, el operador += añade el valor de la derecha a la variable de la izquierda; `x += 42` es lo mismo que `= x + 42`).

4.3. Variables Compuestas

ksh93 introduce una nueva característica, llamada *variables compuestas*. Son similares en naturaleza a un registro de Pascal o Ada o a una estructura de C, y le permiten agrupar elementos relacionados bajo el mismo nombre. He aquí algunos ejemplos:

```
now="May 20 2001 19:44:57"              # Asignar la fecha actual a la variable now
now.hour=19                             # Establecer la hora
now.minute=44                           # Establecer el minuto
```

Observe el uso del punto en el nombre de la variable. Aquí, `now` se denomina variable padre, y debe existir (es decir, tener un valor) antes de que pueda asignar un valor a un componente individual (como `hour` o `minute`). Para acceder a una variable compuesta, debe encerrar el nombre de la variable entre llaves. Si no lo hace, el punto finaliza la búsqueda del nombre de la variable por parte del shell:

```
$ print ${now.hour}
19
$ print $now.hour
May 20 2001 19:44:57.hour
```

4.3.1. Asignación de Variables Compuestas

La asignación a elementos individuales de una variable compuesta es tediosa. En particular, el requisito de que la variable padre exista previamente conduce a un estilo de programación incómodo:

```
person="John Q. Public"
person.firstname=John
person.initial=Q.
person.lastname=Public
```

Afortunadamente, puedes utilizar una asignación compuesta para hacerlo todo de una sola vez:

```
person=(firstname=John initial=Q. lastname=Public
```

Puede recuperar el valor de toda la variable, o de un componente, utilizando *print*.

```
$ print $person                         # Impresion sencilla
( lastname=Public initial=Q. firstname=John )
$ print -r "$person"                    # Impresion a todo su esplendor
(
  lastname=Public
  initial=Q.
  firstname=John
```

```
)
$ print ${person.initial}                # Imprimir solo la inicial del segundo nombre
Q.
```

El segundo comando *print* preserva los espacios en blanco que el shell de Korn proporciona cuando devuelve el valor de una variable compuesta. La opción *-r* para imprimir se discute en el Capítulo 7.

NOTA: El orden de los componentes es diferente del utilizado en la asignación inicial. Este orden depende de cómo el shell de Korn gestiona internamente las variables compuestas y no puede ser controlado por el programador.

Existe una segunda sintaxis de asignación, similar a la primera:

```
person=(typeset firstname=John initial=Q. lastname=Public ;
  typeset -i age=42)
```

Utilizando el comando *typeset*, puede especificar que una variable es un número en lugar de una cadena. Aquí, `person.age` es una variable entera. El resto siguen siendo cadenas. El comando *typeset* y sus opciones se presentan en el Capítulo 6. (También puede utilizar *readonly* para declarar que una variable componente no puede modificarse).

Del mismo modo que puede utilizar += para añadir a una variable normal, también puede añadir componentes a una variable compuesta:

```
person+= (typeset spouse=Jane)
```

Se permite un espacio después del = pero no antes. Esto se aplica a las asignaciones compuestas con = y +=.

El shell de Korn tiene sintaxis adicionales para asignaciones compuestas que se aplican sólo a variables de array; también se discuten en el Capítulo 6.

Finalmente, mencionaremos que el shell de Korn tiene una variable compuesta especial llamada `.sh`. Los diversos componentes se relacionan casi todos con características que aún no hemos cubierto, excepto `$.sh.version`, que le indica la versión del shell de Korn que tiene:

```
$ print ${.sh.version}
Version M 1993-12-28 m
```

Veremos otro componente de `.sh` más adelante en este capítulo, y los otros componentes se cubren a medida que introducimos las características con las que se relacionan.

4.4. Referencias Indirectas a Variables (namerefs)

La mayoría de las veces, como hemos visto hasta ahora, se manipulan variables directamente, por su nombre (x=1, por ejemplo). El shell de Korn le permite manipular variables indirectamente, usando algo llamado *nameref*. Puedes crear un *nameref* usando `typeset -n`, o el más conveniente alias predefinido, *nameref*. He aquí un ejemplo sencillo:

```
$ name="bill"                          # Establecer valor inicial
$ nameref firstname=name               # Configura nameref
$ print $firstname                     # En realidad hace referencia a la variable nameref
bill
$ firstname="arnold"                   # Ahora cambia la referencia indirecta
$ print $name                          # Se cambia la variable original
arnold
```

Para averiguar el nombre de la variable real a la que hace referencia el *nameref*, utilice `$! variable` :

```
$ print ${!firstname}
name
```

A primera vista, esto no parece muy útil. El poder de *namerefs* entra en juego cuando pasas el nombre de una variable a una función, y quieres que esa función pueda actualizar el valor de esa variable. El siguiente ejemplo ilustra cómo funciona:

```
$ date                                 # Dia y horas actuales
Wed May 23 17:49:44 IDT 2001
$ function getday {                    # Definir una funcion
> typeset -n day=$1                    # Configurar el nombre
> day=$(date | awk '{ print $1 }')     # Cambiarlo realmente
> }
$ today=now                            # Establece el valor inicial
$ getday today                         # Ejecutar la funcion
$ print $today                         # Mostrar el valor nuevo
Wed
```

La salida por defecto de *date(1)* tiene este aspecto:

```
$ date
Wed Nov 14 11:52:38 IST 2001
```

La función *getday* utiliza *awk* para imprimir el primer campo, que es el día de la semana. El resultado de esta operación, que se realiza dentro de la sustitución de comandos (descrita más adelante en este capítulo), se asigna a la variable local `day`. Pero `day` es una referencia de nombre; la asignación en realidad actualiza la variable global `today`. Sin la función *nameref*, tendrá que recurrir a trucos avanzados como el uso de *eval* (véase el capítulo 7) para hacer que ocurra algo como esto.

Para eliminar un *nameref*, utilice `unset -n`, que elimina el propio *nameref*, en lugar de eliminar la variable a la que el *nameref* hace referencia. Por último, tenga en cuenta que las variables que son *namerefs* no pueden tener puntos en sus nombres (es decir, ser componentes de una variable compuesta). Sin embargo, pueden ser referencias a una variable compuesta.

4.5. Operadores de Cadena

La sintaxis de llaves permite utilizar los *operadores de cadena* del shell. Los operadores de cadena le permiten manipular valores de variables de varias formas útiles sin tener que escribir programas completos o recurrir a utilidades externas de Unix. Puede hacer mucho con los operadores de manejo de cadenas incluso si todavía no domina las características de programación que veremos en capítulos posteriores.

En particular, los operadores de cadena te permiten hacer lo siguiente:

- Asegurarse de que las variables existen (es decir, están definidas y tienen valores no nulos).
- Establecer valores por defecto para las variables.
- Detectar errores debidos a variables no definidas.
- Eliminar partes de los valores de las variables que coincidan con patrones.

4.5.1. Sintaxis de los Operadores de Cadena

La idea básica detrás de la sintaxis de los operadores de cadena es que los caracteres especiales que denotan operaciones se insertan entre el nombre de la variable y la llave derecha. Cualquier argumento que el operador pueda necesitar se inserta a la derecha del operador.

El primer grupo de operadores de cadena comprueba la existencia de variables y permite sustituirlas por valores por defecto en determinadas condiciones. Se enumeran en la Tabla 4.2.

Tabla 4.2: Operadores de sustitución

Operador	**Sustitución**
`${varname:-word}` **Propósito:** **Ejemplo:**	Si *varname* existe y no es nulo, devuelve su valor; en caso contrario devuelve word. Devolver un valor por defecto si la variable no está definida. `$count:-0` se evalúa a 0 si *count* no está definido.
`${varname:=word}` **Propósito:** **Ejemplo:**	Si *varname* existe y no es nulo, devuelve su valor; en caso contrario, lo establece en word y luego devuelve su valor.[7] Asignar a una variable un valor por defecto si no está definida. `$count:=0` establece el recuento a 0 si no está definido.
`${varname:?word}` **Propósito:** **Ejemplo:**	Si *varname* existe y no es *null*, devuelve su valor; de lo contrario imprime `varname : message`, y aborta el comando o script actual. Omitir mensaje produce el parámetro mensaje por defecto nulo o no establecido. Tenga en cuenta, sin embargo, que los shells interactivos no abortan. Atrapar errores que resultan de variables no definidas. `${count:? ''undefined!''}` imprime `count: ¡undefined!` y sale si count es undefined.
`${varname:+word}` **Objetivo** **Ejemplo:**	Si *varname* existe y no es *null*, devuelve word; en caso contrario devuelve null. Comprobar la existencia de una variable. `${count:+1}` devuelve 1 (que podría significar "verdadero") si *count* está definido.

Los dos puntos (`:`) en cada uno de estos operadores son opcionales. Si se omiten los dos puntos, cambie «existe y no es nulo» por «existe» en cada definición, es decir, el operador sólo comprueba la existencia.

Los dos primeros operadores son ideales para establecer los valores predeterminados de los argumentos de la línea de comandos en caso de que el usuario los omita. De hecho, utilizaremos los cuatro en la Tarea 4-1, que es nuestra primera tarea de programación.

[7] Los programadores de Pascal, Modula y Ada pueden encontrar útil reconocer la similitud de esto con los operadores de asignación en esos lenguajes.

Tarea 4-1

Tienes una gran colección de álbumes y quieres escribir un programa para llevar un registro de ella. Supongamos que tienes un fichero de datos sobre cuántos álbumes tienes de cada artista. Las líneas del fichero tienen este aspecto:

```
14      Bach, J.S.
1       Balachander, S.
21      Beatles
6       Blakey, Art
```

Escriba un programa que imprima las *N* líneas más altas, es decir, los *N* artistas de los que tiene más álbumes. El valor por defecto para *N* debería ser 10. El programa debe tomar un argumento para el nombre del archivo de entrada y un segundo argumento opcional para el número de líneas a imprimir.

Con diferencia, el mejor enfoque para este tipo de script es utilizar las utilidades integradas de Unix, combinándolas con redireccionadores de E/S y tuberías. Esta es la clásica filosofía «building-block» de Unix que es otra de las razones de su gran popularidad entre los programadores. La técnica del bloque de construcción nos permite escribir una primera versión del script de una sola línea:

```
sort -nr "$1" | head -${2:-10}
```

Así es como funciona: el programa *sort(1)* ordena los datos del fichero cuyo nombre se da como primer argumento `($1)`. (Las comillas dobles permiten espacios u otros caracteres inusuales en los nombres de fichero, y también evitan la expansión de comodines). La opción *-n* indica a *sort* que interprete la primera palabra de cada línea como un número (en lugar de como una cadena de caracteres); la opción *-r* le indica que invierta las comparaciones, para ordenar en orden descendente.

La salida de *sort* se envía a la utilidad *head(1)*, que, cuando recibe el argumento *-N*, imprime las primeras N líneas de su entrada en la salida estándar. La expresión `-$2:-10` se evalúa como un guión (-) seguido del segundo argumento, si se da, o como `10` si no se da; observe que la variable en esta expresión es `2`, que es el segundo parámetro posicional.

Supongamos que el script que queremos escribir se llama *highest*. Entonces si el usuario escribe `highest myfile`, la línea que realmente se ejecuta es:

```
sort -nr myfile | head -10
```

O si el usuario escribe highest `myfile 22`, la línea que se ejecuta es:

```
sort -nr myfile | head -22
```

Asegúrese de que entiende cómo el operador de cadena `:-` proporciona un valor por defecto.

Este es un script perfectamente bueno y ejecutable, pero tiene algunos problemas. Primero, su única línea es un poco crítica. Aunque esto no es un gran problema para un script tan pequeño, no es aconsejable escribir scripts largos y elaborados de esta manera. Unos pequeños cambios hacen que el código sea más legible.

Primero, podemos añadir comentarios al código; cualquier cosa entre # y el final de una línea es un comentario. Como mínimo, el script debería empezar con unas pocas líneas de comentario que indiquen lo que hace el script y los argumentos que acepta. A continuación, podemos mejorar los nombres de las variables asignando los valores de los parámetros posicionales a variables regulares con nombres mnemotécnicos. Por último, podemos añadir líneas en blanco para espaciar las cosas; las líneas en blanco, como los comentarios, se ignoran. He aquí una versión más legible:

```
# highest filename [howmany]
#
# Print howmany highest-numbered lines in file filename.
# The input file is assumed to have lines that start with
# numbers.  Default for howmany is 10.

filename=$1

howmany=${2:-10}
sort -nr "$filename" | head -$howmany
```

Los corchetes alrededor de `howmany` en los comentarios se adhieren a la convención en la documentación Unix de que los corchetes denotan argumentos opcionales.

Los cambios que acabamos de hacer mejoran la legibilidad del código, pero no su ejecución. ¿Qué pasaría si el usuario invocara el script sin ningún argumento? Recuerde que los parámetros posicionales por defecto son *null* si no están definidos. Si no hay argumentos, entonces `$1` y `$2` son nulos. La variable howmany (`$2`) está configurada por defecto a 10, pero no hay valor por defecto para `filename ($1)`. El resultado sería que este comando se ejecuta:

```
sort -nr | head -10
```

Si se llama a *sort* sin un argumento de nombre de fichero, se espera que la entrada provenga de la entrada estándar, por ejemplo, una tubería (|) o el teclado del usuario. Como no tiene la tubería, esperará el teclado. Esto significa que el script parecerá colgarse. Aunque siempre puedes teclear CTRL-D o CTRL-C para salir del script, un usuario ingenuo podría no saberlo.

Por lo tanto, tenemos que asegurarnos de que el usuario proporciona al menos un argumento. Hay varias formas de hacerlo; una de ellas implica otro operador de cadena. Reemplazaremos la línea

```
filename=$1
```

Con:

```
filename=${1:?"filename missing."}
```

Esto hace que ocurran dos cosas si un usuario invoca el script sin ningún argumento: en primer lugar, el shell imprime el mensaje un tanto desafortunado en la salida de error estándar:

```
highest: line 1: : filename missing.
```

En segundo lugar, el script sale sin ejecutar el código restante.

Con una modificación algo «chapucera», podemos obtener un mensaje de error ligeramente mejor. Considere este código:

```
filename=$1
filename=${filename:?"missing."}
```

El resultado es el mensaje:

```
highest: line 2: filename: filename missing.
```

(Asegúrese de entender por qué.) Por supuesto, hay formas de imprimir cualquier mensaje que se desee; veremos cómo en el Capítulo 5.

Antes de continuar, examinaremos más detenidamente los dos operadores restantes de la Tabla 4.2 y veremos cómo podemos incorporarlos a la solución de nuestra tarea. El operador := hace aproximadamente lo mismo que :-, excepto que tiene el efecto secundario de establecer el valor de la variable a la palabra dada si la variable no existe.

Por lo tanto, nos gustaría utilizar := en nuestro script en lugar de :-, pero no podemos; estaríamos intentando establecer el valor de un parámetro posicional, lo que no está permitido. Pero si sustituimos:

```
howmany=${2:-10}
```

Con solo:

```
howmany=$2
```

y trasladamos la sustitución a la línea de órdenes real (como hicimos al principio), entonces podríamos utilizar el operador :=:

```
sort -nr "$filename" | head -${howmany:=10}
```

El uso de := tiene la ventaja añadida de fijar el valor de `howmany` en 10 por si lo necesitamos posteriormente en versiones posteriores del script.

El último operador de sustitución es `:+`. Así es como podemos utilizarlo en nuestro ejemplo: digamos que queremos dar al usuario la opción de añadir una línea de cabecera a la salida del script. Si teclea la opción *-h*, la salida irá precedida por la línea:

```
ALBUMS   ARTIST
```

Suponga además que esta opción termina en la variable `header`, es decir, `$header` si la opción *-h* está establecida o nula si no lo está. (Más adelante veremos cómo hacer esto sin alterar los otros parámetros posicionales).

La expresión:

```
${header:+"ALBUMS   ARTIST\n"}
```

produce *null* si la variable *header* es nula o `ALBUMS ARTIST\ n` si no es nula. Esto significa que podemos poner la línea:

```
print -n ${header:+"ALBUMS   ARTIST\n"}
```

justo antes de la línea de comando que hace el trabajo real. La opción *-n* para imprimir hace que no se imprima una nueva línea después de imprimir sus argumentos. Por lo tanto, esta declaración de impresión no imprime nada, ni siquiera una línea en blanco, si `headeres` nulo; de lo contrario, imprime la línea de encabezado y una nueva línea (\ n).

4.5.2. Patrones y Expresiones Regulares

Continuaremos refinando nuestra solución a la Tarea 4-1 más adelante en este capítulo. El siguiente tipo de operador de cadena se utiliza para comparar partes del valor de cadena de una variable con patrones. Los patrones, como vimos en el Capítulo 1, son cadenas que pueden contener caracteres comodín `(*, ? y [ ]` para conjuntos y rangos de caracteres).

Los comodines han sido características estándar de todos los intérpretes de comandos Unix que se remontan (al menos) al intérprete de comandos Thompson de la versión 6.[8] Pero el intérprete de comandos Korn es el primero en añadir capacidades. Añade un conjunto de operadores, llamados *operadores de expresión regular (o regexp para abreviar)*, que le dan

[8] El shell de la versión 6 fue escrito por Ken Thompson. Stephen Bourne escribió el intérprete de comandos Bourne para la versión 7.

mucho del poder de comparación de cadenas de utilidades Unix avanzadas como *awk(1), egrep(1)* (*grep(1)* extendido), y el editor Emacs, aunque con una sintaxis diferente. Estas capacidades van más allá de aquellas a las que puede estar acostumbrado en otras utilidades Unix como *grep, sed(1)* y *vi(1)*.

Los usuarios avanzados de Unix encontrarán útiles las capacidades de expresión regular del shell de Korn para la escritura de scripts, aunque rozan la exageración. (Parte del problema es el inevitable choque sintáctico con la miríada de otros caracteres especiales del shell). Por lo tanto, no entraremos aquí en grandes detalles sobre las expresiones regulares. Para una información más completa, la «última palabra» sobre expresiones regulares prácticas en Unix es *Mastering Regular Expressions*, por Jeffrey E. F. Friedl. Una introducción más suave puede encontrarse en la segunda edición de *sed & awk*, por Dale Dougherty y Arnold Robbins. Ambos están publicados por O'Reilly & Associates. Si ya se siente cómodo con *awk* o *egrep*, puede saltarse la siguiente sección introductoria y pasar a la Sección 4.5.2.3, más adelante en este capítulo, donde explicamos el mecanismo de expresiones regulares del intérprete de órdenes comparándolo con la sintaxis utilizada en esas dos utilidades. De lo contrario, siga leyendo.

Conceptos Básicos de Expresiones Regulares

Piense en las expresiones regulares como cadenas de caracteres que emparejan patrones de forma más potente que el esquema de comodines estándar del shell. Las expresiones regulares surgieron como una idea de la informática teórica, pero han llegado a muchos rincones de la informática práctica cotidiana. La sintaxis utilizada para representarlas puede variar, pero los conceptos son muy parecidos.

Una expresión regular de shell puede contener caracteres regulares, caracteres comodín estándar y operadores adicionales más potentes que los comodines. Cada uno de estos operadores tiene la forma *x(exp)*, donde *x* es el operador concreto y *exp* es cualquier expresión regular (a menudo simplemente una cadena regular). El operador determina cuántas ocurrencias de *exp* puede contener una cadena que coincida con el patrón. La Tabla 4.3 describe los operadores de expresión regular del shell y sus significados.

Tabla 4.3: Operadores de expresiones regulares

Operador	**Significado**
`*(exp)`	Cero ó más ocurrencias de *exp*
`+(exp)`	Uno ó más ocurrencias de *exp*
`?(exp)`	Cero o una ocurrencia de *exp*
`@(exp1|exp2|...)`	Exactamente uno de *exp1* ó *exp2* ó ...
`!(exp)`	Todo lo que no coincida con exp [9]

Como se muestra para el patrón `@( exp1 | exp2 |...)`, un *exp* dentro de cualquiera de los operadores del shell de Korn puede ser una serie de alternativas *exp1|exp2|...*

Una notación alternativa poco conocida es separar cada *exp* con el carácter ampersand, &. En este caso, todas las expresiones alternativas deben coincidir. Piense que | significa «o», mientras que & significa «y». (De hecho, puede utilizar ambos en la misma lista de patrones. El & tiene mayor precedencia, con el significado «coincide con esto y aquello, O coincide con lo siguiente»). La Tabla 4.4 proporciona algunos ejemplos de uso de los operadores de expresiones regulares del shell.

Tabla 4.4: Ejemplos de operadores de expresiones regulares

Expresión	**Coincidencia**
x	x
+(x)	Cadena nula, `x, xx, xxx, ...`
+(x)	`x, xx, xxx, ...`
?(X)	Cadena nula, *x*
!(X)	Cualquier cadena excepto *x*
@(X)	*x* (ver abajo)

Las expresiones regulares son extremadamente útiles cuando se trata de texto arbitrario, como ya sabrá si ha usado *grep* o las capacidades de expresión regular de cualquier editor de Unix. No son tan útiles para hacer coincidir nombres de archivo y otros tipos simples de información con los que los usuarios de shell suelen trabajar. Además, la mayoría de las cosas que puede hacer con los operadores de expresiones regulares del shell también se pueden hacer (aunque posiblemente con más pulsaciones de teclas y menos eficiencia) canalizando la salida de un comando del shell a través de *grep* o *egrep*.

Sin embargo, aquí hay algunos ejemplos de cómo las expresiones regulares de shell pueden resolver problemas de listado de nombres de archivo. Algunos de estos serán útiles en capítulos posteriores como piezas de soluciones para tareas más grandes.

[9] En realidad, !(exp) no es un operador de expresión regular según la definición técnica estándar, aunque es una extensión práctica.

1. El editor de Emacs admite archivos de personalización cuyos nombres terminan en *.el* (para Emacs LISP) o *.elc* (para Emacs LISP compilado). Muestra todos los archivos de personalización de Emacs en el directorio actual.

2. En un directorio de código fuente C, enumere todos los archivos que no son necesarios. Suponga que los archivos «necesarios» terminan en *.co .ho* se denominan *Makefile* o *README*.

3. Los nombres de archivo en el sistema operativo OpenVMS terminan en un punto y coma seguido de un número de versión, por ejemplo, *fred.bob;23*. Muestra todos los nombres de archivo de estilo OpenVMS en el directorio actual.

Aquí están las soluciones:

1. En el primero de ellos, buscamos archivos que terminen en .el con una c opcional . La expresión que coincide con esto es **.el?(c)*.

2. El segundo ejemplo depende de las cuatro subexpresiones estándar **.c, *.h, Makefile* y *README*. La expresión completa es `!(*.c|*.h|Makefile|README)`, que coincide con cualquier cosa que no coincida con ninguna de las cuatro posibilidades.

3. La solución al tercer ejemplo comienza con `*\;`, el comodín de shell *seguido de un punto y coma con barra invertida. Entonces, podríamos usar la expresión regular `+([0-9])`, que coincide con uno o más caracteres en el rango [0-9], es decir, uno o más dígitos. Esto es casi correcto (y probablemente lo suficientemente cerca), pero no tiene en cuenta que el primer dígito no puede ser 0. Por lo tanto, la expresión correcta es `* [1-9]*([0-9])`, que coincide con cualquier cosa que termine con un punto y coma, un dígito del 1 al 9 y cero o más dígitos del 0 al 9.

Adiciones de Clases de Caracteres POSIX

El estándar POSIX formaliza el significado de los caracteres y operadores de expresiones regulares. El estándar define dos clases de expresiones regulares: expresiones regulares básicas (BRE), que son del tipo utilizado por *grep* y *sed* , y expresiones regulares extendidas, que son del tipo utilizado por *egrep* y *awk*.

Para adaptarse a entornos no ingleses, el estándar POSIX mejoró la capacidad de los rangos de juegos de caracteres (p. ej., `[a-z]`) para hacer coincidir caracteres que no están en el alfabeto inglés. Por ejemplo, el è francés es un carácter alfabético, pero la clase de carácter típica `[a-z]` no lo coincidiría. Además, el estándar proporciona secuencias de caracteres

que deben tratarse como una sola unidad al hacer coincidir y cotejar (ordenar) datos de cadena. (Por ejemplo, hay lugares donde los dos caracteres `ch` se tratan como una unidad y deben coincidir y ordenarse de esa manera).

POSIX también cambió lo que había sido la terminología común. Lo que vimos anteriormente en el Capítulo 1 como una «expresión de rango» a menudo se denomina «clase de caracteres» en la literatura de Unix. Ahora se llama una «expresión de paréntesis» en el estándar POSIX. Dentro de las expresiones entre paréntesis, además de los caracteres literales como a, ;, etc., también puede tener componentes adicionales:

Tipo de carácter: Una clase de caracteres POSIX consta de palabras clave entre paréntesis `[: y :]`. Las palabras clave describen diferentes clases de caracteres, como caracteres alfabéticos, caracteres de control, etc. (consulte la Tabla 4.5).

Clasificación de símbolos: Un símbolo de clasificación es una secuencia de varios caracteres que debe tratarse como una unidad. Se compone de los caracteres entre paréntesis `[. y .]`.

Clases de equivalencias: Una clase de equivalencia enumera un conjunto de caracteres que deben considerarse equivalentes, como `e` y `è`. Consiste en un elemento con nombre de la configuración regional, entre paréntesis `[= y =]`.

Las tres construcciones deben aparecer dentro de los corchetes de una expresión entre corchetes. Por ejemplo `[[:alpha:]!]`, coincide con cualquier carácter alfabético o el signo de exclamación; `[[.ch.]]` coincide con el elemento de clasificación, `ch` pero no coincide solo con la letra `c` o la letra `h`. En una configuración regional francesa, `[[=e=]]` puede coincidir con cualquiera de `e, è` o `é`. Las clases y los caracteres coincidentes se muestran en la Tabla 4.5.

Tabla 4.5: Clases de caracteres POSIX

Caracter	Significado
`[:alnum:]`	Caracteres alfanuméricos
`[:alpha:]`	Caracteres alfabéticos
`[:blank:]`	Caracteres de espacios y tabulación
`[:cntrl:]`	Caracteres de control
`[:digit:]`	Caracteres numéricos
`[:graph:]`	Caracteres imprimibles y visibles (sin espacios)
`[:lower:]`	Caracteres en minúsculas
`[:print:]`	Caracteres imprimibles (incluidos los espacios en blanco)
`[:punct:]`	Caracteres de puntuación
`[:space:]`	Caracteres de espacio en blanco
`[:upper:]`	Caracteres en mayúsculas
`[:xdigit:]`	Dígitos hexadecimales

El shell de Korn soporta todas estas características dentro de sus facilidades de concordancia de patrones. Los nombres de clases de caracteres POSIX son los más útiles, porque funcionan en diferentes localizaciones.

La siguiente sección compara las expresiones regulares del shell de Korn con las características análogas de *awk* y *egrep*. Si no está familiarizado con ellas, vaya a la Sección 4.5.3.

Korn Shell Versus Expresiones Regulares awk/egrep

La Tabla 4.6 es una expansión de la Tabla 4.3: la columna del medio muestra los equivalentes en *awk/egrep* de los operadores de expresiones regulares del shell.

Tabla 4.6: Operadores de expresión regular del shell frente a egrep/awk

Korn shell	egrep/awk	Significado
`*(exp)`	`exp*`	Cero o más ocurrencias de *exp*
`+(exp)`	`exp+`	Uno o más ocurrencias de *exp*
`?(exp)`	`exp?`	Cero o una ocurrencia de *exp*
`@(exp1|exp2|...)`	`exp1 | exp2 |...`	*exp1* ó *exp2* ó ...
`!(exp)`	`(none)`	Todo lo que no coincida con exp
`\ N`	`\ N (grep)`	Coincide con el mismo texto que la subexpresión anterior entre paréntesis número N

Estos equivalentes son cercanos pero no del todo exactos. Debido a que el shell interpretaría una expresión `dave|fred|bob` como una canalización de comandos, debe usar `@(dave|fred|bob)` para alternativas por sí mismos.

El comando *grep* tiene una función llamada *backreferences* (o *backrefs*, para abreviar). Esta

función proporciona una abreviatura para repetir partes de una expresión regular como parte de un todo mayor. Funciona de la siguiente manera:

```
grep '\(abc\).*\1' archivo1 archivo2
```

Esto coincide con *abc*, seguido de cualquier número de caracteres, seguido de nuevo por *abc*. De esta forma se pueden referenciar hasta nueve subexpresiones entre paréntesis. El intérprete de comandos Korn ofrece una función análoga. Si utiliza uno o más patrones de expresión regular dentro de un patrón completo, puede hacer referencia a los anteriores utilizando la notación `\ N` como para *grep*.

Por ejemplo:

- `@(dave|fred|bob)` coincide con *dave, fred o bob*.
- `@(*dave*&*fred*)` coincide con *davefred* y *freddave*. (Observe la necesidad de los caracteres *).
- `@(fred)*\1` Coincide con *freddavefred, fredbobfred*, etc.
- `*(dave|fred|bob)` significa «0 o más apariciones de *dave, fred o bob*». Esta expresión coincide con cadenas como la cadena nula, *dave, davedave, fred, bobfred, bobbobdave-fredbobfred*, etc.
- `+(dave|fred|bob)` coincide con cualquiera de las anteriores excepto con la cadena nula.
- `?(dave|fred|bob)` coincide con la cadena nula, *dave, fred o bob*.
- `!(dave|fred|bob)` coincide con cualquier cosa excepto *dave, fred o bob*.

Vale la pena recalcar que las expresiones regulares del shell pueden contener comodines estándar del shell. Así, el comodín del shell `?` (coincide con cualquier carácter) es equivalente a `.` en *egrep* o *awk*, y el operador de conjunto de caracteres del shell `[...]` es el mismo que en esas utilidades.[10] Por ejemplo, la expresión `+([[:dígito:]])` coincide con un número, es decir, uno o más dígitos. El carácter comodín del shell * es equivalente a la expresión regular del shell `*(?)`. Incluso puede anidar las expresiones regulares: `+([[:dígito:]]|!([[:mayúscula:]]))` coincide con uno o más dígitos o letras no mayúsculas.

Dos operadores regexp de x*egrep* y *awk* no tienen equivalentes en el shell de Korn:

[10] Y, para el caso, lo mismo que en *grep, sed, ed, vi,* etc. Una diferencia notable es que el shell utiliza ! dentro de [...] para la negación, mientras que las distintas utilidades utilizan todas ^.

- Los operadores de principio y fin de línea ^y $.
- Los operadores de principio y final de palabra \< y \>.

Apenas son necesarios, ya que el intérprete de comandos Korn no opera normalmente con archivos de texto y analiza las cadenas en palabras por sí mismo. (Esencialmente, ^y $ están implícitos como si siempre estuvieran ahí. Rodee un patrón con caracteres * para desactivar esto). Siga leyendo para conocer más características de la última versión de *ksh*.

Coincidencia de Patrones con Expresiones Regulares

A partir de *ksh93l*, el intérprete de órdenes proporciona una serie de funciones adicionales de expresiones regulares. Las discutimos aquí por separado, porque es muy probable que su versión de *ksh93* no las tenga, a menos que descargue un binario de *ksh93* o construya *ksh93* desde el código fuente. Las facilidades se desglosan de la siguiente manera.

Nuevos operadores de coincidencia de patrones: Hay varias funciones nuevas de coincidencia de patrones disponibles. Se describen brevemente en la Tabla 4.7. Más discusión sigue después de la mesa.

Subpatrones con opciones: Los subpatrones especiales entre paréntesis pueden contener opciones que controlan la coincidencia dentro del subpatrón o el resto de la expresión.

Nueva clase de carácter `:palabra::` La clase de carácter `[:word:]` dentro de una expresión entre paréntesis coincide con cualquier carácter que sea «componente de palabra». Esto es básicamente cualquier carácter alfanumérico o el guión bajo (_).

Secuencias de escape reconocidas dentro de subpatrones: Varias secuencias de escape se reconocen y tratan especialmente dentro de las expresiones entre paréntesis.

Tabla 4.7: Nuevos operadores de coincidencia de patrones en *ksh93l* y versiones posteriores

Operador	Significado
`N(exp)`	Exactamente N ocurrencias de *exp*
`{N,M}(exp)`	Entre N y M ocurrencias de *exp*
`*-(exp)`	Cero o más ocurrencias de *exp* , coincidencia más corta
`+-(exp)`	Una o más ocurrencias de *exp* , coincidencia más corta
`?-(exp)`	Cero o una ocurrencia de *exp* , coincidencia más corta
`@-(exp1|exp2|...)`	Exactamente uno de *exp1* o *exp2* o ..., coincidencia más corta
`{N}-(exp)`	Exactamente N ocurrencias de *exp* , coincidencia más corta
`{N,M}-(exp)`	Entre N y M ocurrencias de *exp* , coincidencia más corta

Los primeros dos operadores en esta tabla coinciden con las facilidades en *egrep (1)*, llamadas *expresiones de intervalo*. Le permiten especificar que desea hacer coincidir exactamente N elementos, ni más ni menos, o que desea hacer coincidir N y M elementos.

El resto de los operadores realizan coincidencias más cortas o «no codiciosas». Normalmente, las expresiones regulares coinciden con el texto más largo posible. Una coincidencia no codiciosa es uno de los textos más cortos posibles que coinciden. El emparejamiento no codicioso se popularizó por primera vez con el lenguaje *perl*. Estos operadores funcionan con los operadores de coincidencia y sustitución de patrones descritos en la siguiente sección; retrasamos los ejemplos de emparejamiento voraz vs. no voraz hasta allí. El comodín de nombre de archivo efectivamente siempre hace una coincidencia codiciosa.

Dentro de operaciones como texttt@(...), puede proporcionar un subpatrón especial que habilite o deshabilite opciones para coincidencias codiciosas e independientes de mayúsculas y minúsculas. Este subpatrón tiene una de las siguientes formas:

```
~(+ options: pattern list)  # Habilitar opciones
~(- options: pattern list)  # Deshabilitar opciones
```

Las opciones son una o ambas *i* para la coincidencia independiente de mayúsculas y minúsculas y *g* para la coincidencia codiciosa. Si `: pattern list` se omite, las opciones se aplican al resto del patrón envolvente. Si se proporcionan, se aplican solo a esa lista de patrones. También es posible omitir las opciones , pero hacerlo realmente no le proporciona ningún valor nuevo.

La expresión entre paréntesis `[[:word:]]` es una abreviatura de `[[:alnum:]_]`. Es una conveniencia notacional, pero que puede aumentar la legibilidad del programa.

Dentro de las expresiones entre paréntesis, *ksh* reconoce todas las secuencias de escape estándar ANSI C y tienen su significado habitual. (Consulte la Sección 7.3.3.1, en el Capítulo 7). Además, las secuencias de escape enumeradas en la Tabla 4.8 se reconocen y se pueden usar para la comparación de patrones.

Tabla 4.8: Secuencias de escape de expresiones regulares

Secuencia de escape	**Significado**
\ d	Igual que `[[:digit:]]`
\ D	Igual que `[![:digit:]]`
\ s	Igual que `[[:space:]]`
\ S	Igual que `[![:space:]]`
\ w	Igual que `[[:word:]]`
\ W	Igual que `[![:word:]]`

¡Uf! Todo esto es bastante embriagador. Si te sientes un poco abrumado, no te preocupes. A medida que aprendas más sobre expresiones regulares y programación en el shell y empieces a realizar tareas de procesamiento de texto cada vez más complejas, llegarás a apreciar el hecho de poder hacer todo esto dentro del propio shell, en lugar de tener que recurrir a programas externos como `sed, awk` o *perl*.

4.5.3. Operadores de Coincidencia de Patrones

La Tabla 4.9 enumera los operadores de concordancia de patrones del shell de Korn.

Tabla 4.9: Operadores de concordancia de patrones

Operador	**Significado**
`${variable#pattern}`	Si el patrón coincide con el comienzo del valor de la variable, elimine la parte más corta que coincida y devuelva el resto.
`${variable##pattern}`	Si el patrón coincide con el comienzo del valor de la variable, elimine la parte más larga que coincida y devuelva el resto.
`${variable%pattern}`	Si el patrón coincide con el final del valor de la variable, elimine la parte más corta que coincida y devuelva el resto.
`${variable%%pattern}`	Si el patrón coincide con el final del valor de la variable, elimine la parte más larga que coincida y devuelva el resto.

Pueden ser difíciles de recordar, así que aquí tienes un práctico recurso mnemotécnico: # corresponde a la parte delantera porque los signos numéricos *preceden* a los números; % corresponde a la parte trasera porque los signos de porcentaje *siguen* a los números. Otro recurso mnemotécnico proviene de la ubicación típica (al menos en EE.UU.) de las teclas # y % en el teclado. La # está a la izquierda y el % a la derecha.

El uso clásico de los operadores de concordancia de patrones es la eliminación de componentes de los nombres de ruta, como prefijos de directorio y sufijos de nombres de archivo. Con esto en mente, aquí hay un ejemplo que muestra cómo funcionan todos los operadores. Supongamos que la variable *path* tiene el valor `/home/billr/mem/long.file.name`; entonces:

Expresión	**Resultado**
`${path##/*/}`	`long.file.name`
`${path#/*/}`	`billr/mem/long.file.name`
`$path`	`/home/billr/mem/long.file.name`

Expresión	Resultado
`${path%.*}`	`/home/billr/mem/long.file`
`${path%%.*}`	`/home/billr/mem/loang`

Los dos patrones utilizados aquí son `/*/`, que coincide con cualquier cosa entre dos barras, y *.**, que coincide con un punto seguido de cualquier cosa.

A partir de *ksh93l*, estos operadores establecen automáticamente la variable de array `.sh.match`. Esto se discute en la Sección 4.5.7, más adelante en este capítulo.

Incorporaremos uno de estos operadores en nuestra próxima tarea de programación, Tarea 4-2.

Tarea 4-2

Estás escribiendo un compilador de C, y quieres usar el shell de Korn para tu front-end.[a]

[a] No te rías: antaño, muchos compiladores de Unix tenían shell scripts como interfaz.

Imagina un compilador de C como una cadena de componentes de procesamiento de datos. El código fuente C se introduce al principio de la tubería, y el código objeto sale al final; hay varios pasos intermedios. La tarea del script de shell, entre otras muchas cosas, es controlar el flujo de datos a través de los componentes y designar los archivos de salida.

Tienes que escribir la parte del script que toma el nombre del fichero fuente C de entrada y crea a partir de él el nombre del fichero de código objeto de salida. Es decir, usted debe tomar un nombre de archivo que termina en `.c` y crear un nombre de archivo que es similar, excepto que termina en `.o`.

La tarea en cuestión es quitar el `.c` del nombre de archivo y añadir `.o`. Una única sentencia shell lo hace:

```
objname=${filename%.c}.o
```

Esto le dice al shell que busque `.c` al final de `filename`. Si hay una coincidencia, devuelve `$filename` con la coincidencia eliminada. Así que si `filename` tenía el valor `fred.c`, la expresión `$filenmame%.c` devolvería `fred`. El `.o` se añade para obtener el `fred.o` deseado, que se almacena en la variable `objname`.

Si `filename` tuviera un valor inapropiado (sin `.c`) como `fred.a`, la expresión anterior se evaluaría como `fred.a.o`: como no hay coincidencia, no se elimina nada del valor de

`filename`, y se añade `.o` de todos modos. Y, si `filename` contuviera más de un punto - por ejemplo, si fuera *y.tab.c* que es tan infame entre los compiladores - la expresión aún produciría el deseado *y.tab.o*. Observe que esto no sería cierto si usáramos `%%` en la expresión en lugar de `%`. El primer operador utiliza la coincidencia más larga en lugar de la más corta, por lo que coincidiría con `.tab.o` y se evaluaría a `y.o` en lugar de `y.tab.o`. Así que el `%` simple es correcto en este caso.

Sin embargo, para la Tarea 4-3 sería preferible un borrado de coincidencia más larga.

Tarea 4-3

Está implementando un filtro que prepara un archivo de texto para su impresión. Quiere poner el nombre del fichero - sin ningún prefijo de directorio - en la página «banner». Suponga que, en su script, tiene la ruta del archivo a imprimir almacenada en la variable `pathname`.

Está claro que el objetivo es eliminar el prefijo del directorio de la ruta. La siguiente línea lo hace:

```
bannername=${pathname##*/}
```

Esta solución es similar a la primera línea de los ejemplos mostrados anteriormente. Si *pathname* fuera sólo un nombre de fichero, el patrón `*/` (cualquier cosa seguida de una barra) no coincidiría, y el valor de la expresión sería `$pathname` sin modificar. Si *pathname* fuera algo como `fred/bob`, el prefijo `fred/` coincidiría con el patrón y se eliminaría, dejando sólo `bob` como valor de la expresión. Lo mismo ocurriría si la ruta fuera algo como `/dave/pete/fred/bob`: como el ## borra la coincidencia más larga, borra todo `/dave/pete/fred/`.

Si usáramos `#*/` en lugar de `##*/`, la expresión tendría el valor incorrecto `dave/pete/fred/bob`, porque la instancia más corta de «cualquier cosa seguida de una barra» al principio de la cadena es sólo una barra (/).

La construcción `${variable##*/}` es bastante similar a la utilidad de Unix *basename(1)*. *basename* es menos eficiente que `${variable##/*}` porque puede ejecutarse en su propio proceso separado en lugar de dentro del shell.[11] Otra utilidad, *dirname(1)*, hace esencialmente lo contrario que *basename*: devuelve sólo el prefijo del directorio. Es equivalente a la expresión del shell de Korn `${variable%/*}` y es menos eficiente por la misma razón.

[11] basename puede estar incorporado en algunas versiones de *ksh93*. Por lo tanto, no se garantiza que se ejecute en un proceso separado.

4.5.4. Operadores de Sustitución de Patrones

Además de los operadores de concordancia de patrones que eliminan fragmentos de los valores de las variables del shell, puedes hacer sustituciones en esos valores, como en un editor de texto. (De hecho, usando estas facilidades, casi podría escribir un editor de texto en modo línea como un script del shell). Estos operadores están listados en la Tabla 4.11.

Tabla 4.11: Operadores de sustitución de patrones

Operación	**Significado**
`${variable:start}`	Representan operaciones de subcadena. El resultado es el valor de la variable que empieza en la posición inicial y sigue por los caracteres de longitud. El primer carácter se sitúa en la posición 0 y, si no se indica ninguna longitud, se utiliza el resto de la cadena. Cuando se utiliza con $* o $@ o una matriz indexada por * o @ (véase el Capítulo 6), *start* es un índice inicial y *length* es el recuento de elementos. En otras palabras, el resultado es una porción de los parámetros posicionales o de la matriz. Tanto *start* como *length* pueden ser expresiones aritméticas. A partir de *ksh93m*, un inicio negativo se toma como relativo al final de la cadena. Por ejemplo, si una cadena tiene 10 caracteres, numerados del 0 al 9, un valor de inicio de -2 significa 7 (9 - 2 = 7). Del mismo modo, si la variable es una matriz indexada, un inicio negativo produce un índice trabajando hacia atrás desde el subíndice más alto de la matriz.
`${variable:start:length}`	
`${variable/pattern/replace}`	Si la variable contiene una coincidencia de patrón, la primera coincidencia se sustituye por el texto de reemplazar.
`${variable//pattern/replace}`	Es la misma operación que la anterior, salvo que se sustituyen todas las coincidencias del patrón.

Operación	Significado
`${variable/pattern}`	Si la variable contiene una coincidencia de patrón, elimina la primera coincidencia de patrón.
`${variable/#pattern/replace}`	Si la variable contiene una coincidencia de patrón, la primera coincidencia se sustituye por el texto de sustitución. La coincidencia debe producirse al principio del valor de la variable. Si no coincide allí, no se produce ninguna sustitución.
`${variable/%pattern/replace}`	Si la variable contiene una coincidencia de patrón, la primera coincidencia se sustituye por el texto de sustitución. La coincidencia debe producirse al final del valor de la variable. Si no coincide allí, no se produce ninguna sustitución.

La sintaxis `${variable/pattern}` es diferente de los operadores #, ##, % y %% que vimos anteriormente. Esos operadores están limitados a coincidir al principio o al final del valor de la variable, mientras que la sintaxis que se muestra aquí no lo está. Por ejemplo:

```
$ path=/home/fred/work/file
$ print ${path/work/play}              # Reemplaza work por play
/home/fred/play/file
```

Volvamos a nuestro ejemplo del front-end del compilador y veamos cómo podríamos utilizar estos operadores. Al convertir un nombre de archivo fuente C en un nombre de archivo objeto, podríamos hacer la sustitución de esta manera:

```
objname=${filename/%.c/.o}             # Cambia .c por .o, pero solo al final
```

Si tuviéramos una lista de nombres de archivo C y quisiéramos cambiarlos todos por nombres de archivo objeto, podríamos utilizar el llamado operador de sustitución global:

```
llfiles="fred.c dave.c pete.c"
$ allobs=${allfiles//.c/.o}
$ print $allobs
fred.o dave.o pete.o
```

Los patrones pueden ser cualquier expresión de patrón del shell de Korn, como se ha comentado anteriormente, y el texto de sustitución puede incluir la notación \ N para obtener el texto que coincida con un sub-patrón.

Finalmente, estas operaciones pueden aplicarse a los parámetros posicionales y a los arrays,

en cuyo caso se realizan sobre todos los parámetros o elementos del array a la vez. (Las matrices se describen en el capítulo 6.)

```
$ print "$@"
hi how are you over there
$ print ${@/h/H}
Hi How are you over tHere             # Cambia h por H en todos los parametros
```

Emparejamiento codicioso versus no codicioso

Como prometí, aquí hay una breve demostración de las diferencias entre las expresiones regulares con y sin codicia:

```
$ x='12345abc6789'
$ print ${x//+([[:digit:]])/X}            # Sustitucion con la coincidencia mas larga
XabcX
$ print ${x//+-([[:digit:]])/X}           # Sustitucion con coincidencia mas corta
XXXXXabcXXXX
$ print ${x##+([[:digit:]])}              # Sustitucion con la coincidencia mas larga
abc6789
$ print ${x#+([[:digit:]])}               # Eliminar la coincidencia mas corta
2345abc6789
```

La primera impresión sustituye la coincidencia más larga de «uno o más dígitos» por una sola X, en todas partes de la cadena. Como se trata de la coincidencia más larga, se sustituyen ambos grupos de dígitos. En el segundo caso, la coincidencia más corta para «uno o más dígitos» es un solo dígito, por lo que cada dígito se sustituye por una X.

Del mismo modo, los casos tercero y cuarto demuestran la eliminación de texto de la parte delantera del valor, utilizando la coincidencia más larga y la más corta. En el tercer caso, la coincidencia más larga elimina todos los dígitos; en el cuarto caso, la coincidencia más corta elimina un solo dígito.

4.5.5. Operadores de nombre de variable

Varios operadores se relacionan con nombres de variables del shell, como se ve en la Tabla 4.12.

Tabla 4.12: Operadores relacionados con el nombre

Operador	Significado
`${!variable}`	Devuelve el nombre de la variable real referenciada por la variable nameref.
`${!base*}` `${!base@}`	Lista de todas las variables cuyos nombres empiezan por base.

Las referencias a nombres se trataron en la Sección 4.4, anteriormente en este capítulo. Véase allí un ejemplo de `${!name}`.

Los dos últimos operadores de la Tabla 4.12 pueden ser útiles para depurar y/o rastrear el uso de variables en un script grande. Sólo para ver cómo funcionan:

```
$ print ${!HIST*}
HISTFILE HISTCMD HISTSIZE
$ print ${!HIST@}
HISTFILE HISTCMD HISTSIZE
```

En el capítulo 6 se describen otros operadores relacionados con las variables de matriz.

4.5.6. Operadores de longitud

Hay tres operadores restantes sobre variables. Uno es `${#varname}`, que devuelve el número de caracteres de la cadena.[12] (En el Capítulo 6 veremos cómo tratar éste y otros valores similares como números reales para poder usarlos en expresiones aritméticas). Por ejemplo, si *filename* tiene el valor *fred.c*, entonces `${#filename}` tendría el valor 6. Los otros dos operadores (`${#array[*]}` y `${#array[@]}`) tienen que ver con las variables de array, que también se tratan en el Capítulo 6.

4.5.7. La variable *.sh.match*

La variable `.sh.match` se introdujo en *ksh93l*. Es una matriz indexada (véase el Capítulo 6), cuyos valores se establecen cada vez que se realiza una operación de coincidencia de patrones en una variable, como `${filename%%*/}`, con cualquiera de los operadores #, % (para la coincidencia más corta), o ##, %% (para la coincidencia más larga), o / y // (para sustituciones). `.sh.match[0]` contiene el texto que coincide con el patrón completo. `.sh.match[1]` contiene el texto que coincide con la primera subexpresión entre paréntesis, `.sh.match[2]` el texto que coincide con la segunda, y así sucesivamente. Los valores de `.sh.match` dejan de ser válidos (es decir, no intente utilizarlos) si cambia la variable sobre la que se realizó la comparación de patrones.

De nuevo, esta es una característica pensada para programación y procesamiento de texto más avanzados, análoga a características similares en otros lenguajes como *perl*. Si estás empezando, no te preocupes.

[12] Puede ser mayor que el número de bytes para juegos de caracteres multibyte.

4.6. Sustitución de comandos

Hasta ahora, hemos visto dos formas de introducir valores en las variables: mediante sentencias de asignación y mediante el suministro por parte del usuario de argumentos en la línea de comandos (parámetros posicionales). Existe otra forma: la sustitución de comandos, que permite utilizar la salida estándar de un comando como si fuera el valor de una variable. Pronto verá lo potente que es esta función.

La sintaxis de la sustitución de comandos es:

```
$(Unix command)
```

El comando dentro del paréntesis se ejecuta, y cualquier cosa que el comando escriba en la salida estándar (y en el error estándar) se devuelve como valor de la expresión. Estas construcciones pueden anidarse, es decir, el comando Unix puede contener sustituciones de comandos.

He aquí algunos ejemplos sencillos:

- El valor de `$(pwd)` es el directorio actual (igual que la variable de entorno `$PWD`).
- El valor de `$(ls)` son los nombres de todos los archivos del directorio actual, separados por nuevas líneas.
- Para encontrar información detallada sobre un comando si no sabe dónde reside su archivo, escriba `ls -l $(whence -p command )`. La opción -p fuerza a whence a hacer una búsqueda por ruta y no considerar palabras clave, built-ins, etc.
- Para obtener el contenido de un archivo en una variable, puede utilizar `varname=$(<filename)`. `$(cat filename)` hará lo mismo, pero el intérprete de comandos utiliza el primero como una abreviatura incorporada y lo ejecuta de forma más eficiente.
- Si quieres editar (con Emacs) todos los capítulos de tu libro en el shell de Korn que tengan la frase «sustitución de comandos», suponiendo que todos tus archivos de capítulos empiecen por ch, podrías escribir:

  ```
  emacs $(grep -l 'command substitution' ch*.xml)
  ```

  La opción `-l` de grep imprime sólo los nombres de los archivos que contienen coincidencias.

La sustitución de órdenes, al igual que la expansión de variables, se realiza entre comillas

dobles. (Las comillas dobles dentro de la sustitución de comandos no se ven afectadas por las comillas dobles que las encierran). Por lo tanto, nuestra regla en el Capítulo 1 y el Capítulo 3 sobre el uso de comillas simples para cadenas a menos que contengan variables se ampliará ahora: «En caso de duda, use comillas simples, a menos que la cadena contenga variables o sustituciones de comandos, en cuyo caso use comillas dobles».

(Por compatibilidad con versiones anteriores, el shell de Korn soporta la notación original de sustitución de comandos del shell de Bourne (y del shell C) usando comillas inversas: `` `...` ``. Sin embargo, es considerablemente más difícil de usar que `$(...)`, ya que las comillas y las sustituciones de órdenes anidadas requieren un cuidadoso escape. No utilizamos las comillas traseras en ninguno de los programas de este libro).

Sin duda se le ocurrirán muchas formas de usar la sustitución de comandos a medida que gane experiencia con el shell de Korn. Una que es un poco más compleja que las mencionadas anteriormente se relaciona con una tarea de personalización que vimos en el Capítulo 3: personalizar su cadena del prompt.

Recuerde que puede personalizar su cadena del prompt asignando un valor a la variable `PS1`. Si estás en una red de ordenadores, y usas diferentes máquinas de vez en cuando, puede resultarte útil tener el nombre de la máquina en la que estás en tu cadena del prompt. La mayoría de las versiones modernas de Unix tienen el comando *hostname(1)*, que imprime el nombre de red de la máquina en la que se encuentra en la salida estándar. (Si usted no tiene este comando, puede tener uno similar como *uname*.) Este comando le permite obtener el nombre de la máquina en su cadena del prompt poniendo una línea como esta en su *.profile* o archivo de entorno:

```
PS1="$(hostname) $ "
```

(En este caso, no es necesario que el segundo signo de dólar vaya precedido de una barra invertida. Si el carácter después del $ no es especial para el shell, el $ se incluye literalmente en la cadena). Por ejemplo, si su máquina tuviera el nombre `coltrane`, entonces esta sentencia establecería su cadena prompt a `coltrane $ "`.

La sustitución de comandos nos ayuda con la solución de la siguiente tarea de programación, Tarea 4-4, relacionada con la base de datos de álbumes de la Tarea 4-1.

Tarea 4-4

El fichero utilizado en la Tarea 4-1 es en realidad un informe derivado de una tabla más grande de datos sobre álbumes. Esta tabla consta de varias columnas, o campos, a los que el usuario se refiere con nombres como «artist», «title», «year», etc. Las columnas están separadas por barras verticales (|, igual que el carácter pipa de Unix). Para tratar columnas individuales en la tabla, los nombres de campo deben convertirse en números de campo.

Supongamos que existe una función del shell llamada *getfield* que toma el nombre del campo como argumento y escribe el número de campo correspondiente en la salida estándar. Utilice esta rutina para extraer una columna de la tabla de datos.

La utilidad *cut(1)* es ideal para esta tarea. *cut* es un filtro de datos: extrae columnas de datos tabulares.[13] Si proporciona el número de columnas que desea extraer de la entrada, *cut* imprime sólo esas columnas en la salida estándar. Las columnas pueden ser posiciones de caracteres o - relevante en este ejemplo - campos que están separados por caracteres TAB u otros delimitadores.

Supongamos que la tabla de datos de nuestra tarea es un fichero llamado *albums* y que tiene el siguiente aspecto:

```
Coltrane, John|Giant Steps|Atlantic|1960|Ja
Coltrane, John|Coltrane Jazz|Atlantic|1960|Ja
Coltrane, John|My Favorite Things|Atlantic|1961|Ja
Coltrane, John|Coltrane Plays the Blues|Atlantic|1961|Ja
...
```

Así es como utilizaríamos *cut* para extraer la cuarta columna (año):

```
cut -f4 -d\| albums
```

El argumento *-d* se utiliza para especificar el carácter utilizado como delimitador de campo (TAB es el predeterminado). La barra vertical debe tener barra diagonal inversa para que el shell no intente interpretarla como una tubería.

A partir de esta línea de código y de la rutina *getfield*, podemos deducir fácilmente la solución a la tarea. Supongamos que el primer argumento de *getfield* es el nombre del campo que el usuario quiere extraer. Entonces la solución es:

```
fieldname=$1
cut -f$(getfield $fieldname) -d\| albums
```

[13] Algunos sistemas muy antiguos derivados de BSD no tienen *cut*, pero puedes usar *awk* en su lugar. Siempre que veas un comando de la forma textttcut -f N -d C nombrearchivo, usa esto en su lugar: `awk -F C '{print $ N}' filename`.

Si ejecutáramos este script con el argumento `year`, la salida sería:

```
1960
1960
1961
1961
...
```

La Tarea 4-5 es otra pequeña tarea que utiliza `cut`.

Tarea 4-5

Suponga que ha iniciado sesión en un gran servidor o mainframe que admite muchos usuarios simultáneos. Envíe un mensaje de correo electrónico a todas las personas que estén conectadas en ese momento.

El comando *who(1)* te dice quién está conectado (así como en qué terminal está y cuándo se conectó). Su salida tiene este aspecto:

```
billr       console      May 22 07:57
fred        tty02        May 22 08:31
bob         tty04        May 22 08:12
```

Los campos están separados por espacios, no por TABs. Como necesitamos el primer campo, podemos utilizar un espacio como separador de campos en el comando de *cut*. (De lo contrario, tendríamos que utilizar la opción de *cut* que utiliza columnas de caracteres en lugar de campos). Para proporcionar un carácter de espacio como argumento en una línea de comandos, puede rodearlo de comillas:

```
who | cut -d' ' -f1
```

Con la salida de who anterior, la salida de este comando se vería así:

```
billr
fred
bob
```

Esto lleva directamente a la solución de la tarea. Sólo tienes que escribir:

```
mail $(who | cut -d' '  -f1)
```

Se ejecutará el comando `mail billr fred bob` y entonces podrás escribir tu mensaje.

La Tarea 4-6 es otra tarea que muestra lo útiles que pueden ser las canalizaciones de comandos en la sustitución de comandos.

Tarea 4-6

El comando *ls* permite buscar patrones con comodines, pero no permite seleccionar archivos por fecha de modificación. Diseña un mecanismo que te permita hacerlo.

Esta tarea se inspiró en la función del sistema operativo OpenVMS que permite especificar archivos por fecha con los parámetros *BEFORE* y *SINCE*.

Aquí tienes una función que te permite listar todos los ficheros que fueron modificados por última vez en la fecha que des como argumento. Una vez más, elegimos una función por razones de velocidad. El nombre de la función no pretende hacer ningún juego de palabras:

```
function lsd {
  date=$1
  ls -l | grep -i "^.\{41\}$date" | cut -c55-
}
```

Esta función depende de la disposición de las columnas del comando *ls -l*. En particular, depende de que las fechas empiecen en la columna 42 y los nombres de fichero empiecen en la columna 55. Si este no es el caso en tu versión de Unix, necesitarás ajustar los números de columna [14].

Usamos la utilidad de búsqueda *grep* para hacer coincidir la fecha dada como argumento (en la forma *Mon DD*, por ejemplo, `Jan 15` u `Oct 6`, este último con dos espacios) con la salida de `ls -l`. (El argumento de expresión regular para *grep* se entrecomilla con comillas dobles, para realizar la sustitución de variables). Esto nos da un largo listado de sólo aquellos ficheros cuyas fechas coinciden con el argumento. La opción *-i* de *grep* le permite utilizar todas las letras minúsculas en el nombre del mes, mientras que el argumento bastante extravagante significa: «Coincidir con cualquier línea que contenga 41 caracteres seguidos del argumento de la función». Por ejemplo, si se escribe `lsd 'jan 15'`, *grep* buscará las líneas que contengan 41 caracteres seguidos de `jan 15` (o `Jan 15`).

La salida de *grep* se canaliza a través de nuestro omnipresente amigo *cut* para recuperar sólo los nombres de archivo. El argumento para *cut* le dice que extraiga los caracteres de la columna 55 hasta el final de la línea.

Con la sustitución de comandos, puede utilizar esta función con cualquier comando que acepte argumentos de nombre de archivo. Por ejemplo, si desea imprimir todos los archivos de su directorio actual que fueron modificados por última vez hoy, y hoy es 15 de enero,

[14] Por ejemplo, `ls -l` en GNU/Linux tiene fechas que comienzan en la columna 43 y nombres de archivo que comienzan en la columna 57.

podría escribir:

```
lp $(lsd 'jan 15')
```

La salida de *lsd* está en múltiples líneas (una por cada nombre de fichero), pero como la variable IFS (ver antes en este capítulo) contiene newline por defecto, el shell usa newline para separar palabras en la salida de /emphlsd, igual que hace normalmente con el espacio y TAB.

4.7. Ejemplos avanzados: pushd y popd

Concluimos este capítulo con un par de funciones que puedes encontrar útiles en tu uso diario de Unix. Resuelven el problema presentado por la Tarea 4-7.

Tarea 4-7

En el shell C, los comandos *pushd* y *popd* implementan una pila de directorios que le permiten moverse a otro directorio temporalmente y que el shell recuerde dónde estaba. El comando *dirs* imprime la pila. El shell de Korn no proporciona estos comandos. Impleméntelos como funciones del shell.

Comenzamos implementando un subconjunto significativo de sus capacidades y terminamos la implementación en el Capítulo 6. (Para facilitar el desarrollo y la explicación, nuestra implementación ignora algunas cosas que una versión más a prueba de balas debería manejar. Por ejemplo, los espacios en los nombres de archivo harán que las cosas se rompan).

Si no sabe lo que es una pila, piense en un receptáculo de platos accionado por un muelle en una cafetería. Al colocar los platos en el recipiente, el muelle se comprime para que la parte superior se mantenga más o menos al mismo nivel. El plato colocado más recientemente en la pila es el primero que se coge cuando alguien quiere comida; por eso, la pila se conoce como una estructura de «último en entrar, primero en salir» o *LIFO*. (Las víctimas de una recesión o de adquisiciones de empresas también reconocerán este mecanismo en el contexto de las políticas de despido de las empresas). Poner algo en una pila se conoce en informática como *pushing*, y quitar algo de la parte superior se llama *popping*.

Una pila es muy útil para recordar directorios, como veremos; puede «mantener su lugar» hasta un número arbitrario de veces. La forma `cd -` del comando *cd* hace esto, pero sólo a un nivel. Por ejemplo: si estás en *firstdir* y luego cambias a *seconddir*, puedes escribir `cd -` para volver. Pero si empiezas en *firstdir*, luego cambias a *seconddir*, y luego vas a

thirddir, puedes usar `cd -` sólo para volver a *seconddir*. Si escribes `cd -` de nuevo, volverás a *thirddir*, porque es el directorio anterior[15].

Si quieres la funcionalidad «anidada» de recordar-y-cambiar que te llevará de vuelta a *firstdir*, necesitas una pila de directorios junto con los comandos *dirs, pushd* y *popd*. Así es como funcionan:[16]

- `pushd dir` hace un `cd` a `dir` y luego empuja *dir* a la pila.
- *popd* hace un `cd` al directorio superior, luego lo saca de la pila.

Por ejemplo, considera la serie de eventos de la Tabla 4-12. Suponga que acaba de iniciar sesión y que se encuentra en su directorio personal (*/home/you*).

Implementaremos una pila como una variable de entorno que contiene una lista de directorios separados por espacios.

Tabla 4.13: Ejemplo de pushd/popd

Comando	**Contenido de la pila (arriba a la izquierda)**	**Directorio de resultados**
`pushd fred`	`/home/you/fred`	`/home/you/fred`
`pushd /etc`	`/etc /home/you/fred`	`/etc`
`cd /usr/tmp`	`/etc /home/you/fred`	`/usr/tmp`
`popd`	`/home/you/fred`	`/etc`
`popd`	`(empty)`	`/home/you/fred`

Tu pila de directorios debería inicializarse en tu directorio personal cuando te conectes. Para ello, pon esto en tu *.profile*:

```
DIRSTACK="$PWD"
export DIRSTACK
```

No ponga esto en su fichero de entorno si tiene uno. La sentencia *export* garantiza que `DIRSTACK` es conocido por todos los subprocesos; usted quiere inicializarlo sólo una vez. Si pone este código en un fichero de entorno, se reiniciará en cada subproceso del shell interactivo, lo que probablemente no quiera.

A continuación, necesitamos implementar *dirs, pushd* y *popd* como funciones. Aquí están nuestras versiones iniciales:

```
function dirs {     # imprimir pila de directorios (facil)
```

[15] Piense en `cd -` como sinónimo de `cd $OLDPWD`; véase el capítulo anterior.

[16] Aquí lo hemos hecho de forma diferente al shell C. El shell C *pushd* empuja primero el directorio inicial a la pila, seguido del argumento del comando. El shell C *popd* elimina el directorio superior de la pila, revelando un nuevo directorio superior. A continuación, *cds* al nuevo directorio superior. Creemos que este comportamiento es menos intuitivo que nuestro diseño aquí.

```
  print $DIRSTACK
}

function pushd {    # empujar el directorio actual a la pila
  dirname=$1
  cd ${dirname:?"missing directory name."}
  DIRSTACK="$PWD $DIRSTACK"
  print "$DIRSTACK"
}

function popd {     # cd a top, sacarlo de la pila
  top=${DIRSTACK%% *}
  DIRSTACK=${DIRSTACK#* }
  cd $top
  print "$PWD"
}
```

Observa que no hay mucho código. Repasemos las funciones y veamos cómo funcionan. *dirs* es fácil; sólo imprime la pila. La diversión comienza con *pushd*. La primera línea simplemente guarda el primer argumento en la variable *dirname* por razones de legibilidad.

El propósito principal de la segunda línea es cambiar al nuevo directorio. Usamos el operador `:?` para manejar el error cuando falta el argumento: si el argumento es dado, la expresión `${dirname:? "missing directory name."}` se evalúa a `$dirname`, pero si no es dado, el shell imprime el mensaje `ksh: pushd: line 2: dirname: missing directory name`. y sale de la función.

La tercera línea de la función introduce el nuevo directorio en la pila. La expresión entre comillas dobles consiste en el nombre completo del directorio actual, seguido de un espacio, seguido del contenido de la pila de directorios (`$DIRSTACK`). Las comillas dobles aseguran que todo esto se empaqueta en una sola cadena para asignar de nuevo a `DIRSTACK`.

La última línea simplemente imprime el contenido de la pila, con la implicación de que el directorio situado más a la izquierda es el directorio actual y el primero de la pila. (Esta es la razón por la que elegimos espacios para separar los directorios, en lugar de los más habituales dos puntos como en `PATH` y `MAILPATH`).

La función *popd* hace otro uso de los operadores de coincidencia de patrones del shell. La primera línea utiliza el operador %%, que elimina la coincidencia más larga de « *» (un espacio seguido de cualquier cosa). Esto elimina todo menos la parte superior de la pila. El resultado se guarda en la variable `top`, de nuevo por razones de legibilidad.

La segunda línea es similar, pero va en la otra dirección. Utiliza el operador #, que intenta

borrar la coincidencia más corta del patrón «* » (cualquier cosa seguida de un espacio) del valor de `DIRSTACK`. El resultado es que el directorio superior (y el espacio que le sigue) se borra de la pila.

La tercera línea en realidad cambia el directorio a la parte superior anterior de la pila. (Tenga en cuenta que a *popd* no le importa dónde se encuentre cuando lo ejecute; si su directorio actual es el que está en la parte superior de la pila, no irá a ninguna parte). La línea final sólo imprime un mensaje de confirmación.

Este código es deficiente en los siguientes aspectos: en primer lugar, no prevé errores. Por ejemplo:

- ¿Qué ocurre si el usuario intenta introducir un directorio que no existe o no es válido?
- ¿Qué pasa si el usuario intenta hacer *popd* y la pila está vacía?

Pon a prueba tu comprensión del código averiguando cómo respondería a estas condiciones de error. La segunda deficiencia es que el código implementa sólo algunas de las funcionalidades de los comandos *pushd* y *popd* del shell C - aunque las partes más útiles. En el próximo capítulo, veremos cómo superar estas dos deficiencias.

El tercer problema con el código es que no funcionará si, por alguna razón, un nombre de directorio contiene un espacio. El código tratará el espacio como un carácter separador. Aceptaremos esta deficiencia por ahora. Sin embargo, cuando lea sobre arrays en el Capítulo 6, piense en cómo podría usarlos para reescribir este código y eliminar el problema.

CAPÍTULO 5

CONTROL DE FLUJO

Si es programador, puede que haya leído el último capítulo – con su afirmación al principio de que el shell de Korn tiene un conjunto avanzado de capacidades de programación – y se haya preguntado dónde están muchas características de los lenguajes convencionales. Tal vez el «agujero» más obvio en nuestra cobertura hasta ahora se refiere a las construcciones de *control de flujo* como `if, for, while,` etcétera.

El control de flujo da a un programador el poder de especificar que sólo se ejecuten ciertas partes de un programa, o que ciertas partes se ejecuten repetidamente, según condiciones como los valores de las variables, si los comandos se ejecutan correctamente o no, y otras. Llamamos a esto la capacidad de controlar el flujo de ejecución de un programa.

Casi todos los scripts o funciones de shell mostrados hasta ahora no han tenido control de flujo – ¡sólo han sido listas de comandos a ejecutar! Sin embargo, el shell de Korn, como los shells C y Bourne, tiene todas las capacidades de control de flujo que cabría esperar y más; las examinaremos en este capítulo. Las usaremos para mejorar las soluciones a algunas de las tareas de programación que vimos en el último capítulo y para resolver tareas que introducimos aquí.

Aunque hemos intentado explicar el control de flujo para que los no programadores puedan entenderlo, también simpatizamos con los programadores que temen tener que pasar por otra explicación tabula rasa. Por esta razón, algunas de nuestras discusiones relacionan los mecanismos de control de flujo del shell de Korn con aquellos que los programadores ya deberían conocer. Por lo tanto, estará en una mejor posición para entender este capítulo si ya tiene un conocimiento básico de los conceptos de control de flujo.

El shell de Korn soporta las siguientes construcciones de control de flujo:

- `if/else` Ejecuta una lista de sentencias si una determinada condición es/no es ver-

dadera.

- `for` Ejecuta una lista de sentencias un número fijo de veces.
- `while` Ejecuta una lista de sentencias repetidamente mientras se cumple una determinada condición.
- `until` Ejecuta una lista de sentencias repetidamente hasta que se cumpla una determinada condición.
- `case` Ejecuta una de varias listas de sentencias en función del valor de una variable.

Además, el shell de Korn proporciona un nuevo tipo de construcción de control de flujo:

- `select` Permitir al usuario seleccionar una de una lista de posibilidades de un menú.

Cubriremos cada uno de ellos, pero le advertimos: la sintaxis es inusual.

5.1. if/else

El tipo más simple de construcción de control de flujo es el condicional, encarnado en la sentencia *if* del shell de Korn. Se usa un condicional cuando se quiere elegir si hacer o no hacer algo, o elegir entre un pequeño número de cosas a hacer, de acuerdo con la verdad o falsedad de las condiciones. Las condiciones comprueban los valores de las variables del shell, las características de los archivos, si los comandos se ejecutan correctamente o no, y otros factores. El shell tiene un gran conjunto de pruebas incorporadas que son relevantes para la tarea de programación del shell.

La construcción *if* tiene la siguiente sintaxis:

```
if <condicion>
then
  <declaraciones>
[elif <condiciones>
  then <declaraciones...>
[else
  <declaraciones>
fi
```

La forma más sencilla (sin las partes `elif` y `else`, también conocidas como cláusulas) ejecuta las *sentencias* sólo si la *condición* es verdadera. Si añade una cláusula `else`, podrá ejecutar un conjunto de sentencias si la condición es verdadera u otro conjunto de sentencias si la condición es falsa. Puede utilizar tantas cláusulas `elif` (contracción de *else if*) como desee; introducen más condiciones y, por lo tanto, más opciones para el conjunto

de sentencias a ejecutar. Si utiliza una o más cláusulas `elif`, puede considerar la cláusula `else` como la parte «si todo lo demás falla».

5.1.1. Estado de salida y retorno

Quizá el único aspecto de esta sintaxis que difiere de la de lenguajes convencionales como C y Pascal es que la «condición» es en realidad una lista de sentencias en lugar de la expresión booleana (verdadero o falso) más habitual. ¿Cómo se determina la veracidad o falsedad de la condición? Tiene que ver con un concepto general de Unix que aún no hemos tratado: el *estado de salida* de los comandos.

Cada comando Unix, ya sea que provenga del código fuente en C, algún otro lenguaje, o un script/función del shell, devuelve un código entero al proceso que lo llama – el shell en este caso – cuando termina. Esto se denomina estado de salida. 0 es *usualmente* el estado de salida «OK», mientras que cualquier otro (1 a 255) *usualmente* denota un error.[1] La forma en que *ksh* maneja los estados de salida de los comandos incorporados se describe con más detalle más adelante en esta sección.

`if` comprueba el estado de salida de la *última* sentencia de la lista que sigue a la palabra clave `if`.[2] (Si el estado es 0, la condición se evalúa como verdadera; si es cualquier otra cosa, la condición se considera falsa. Lo mismo ocurre con cada condición adjunta a una sentencia `elif` (si la hay).

Esto nos permite escribir código de la forma:

```
if <comando> es ejecutado con exito
then
  procesamiento normal
else
  procesamiento de errores
fi
```

[1] Dado que se trata de una «convención» y no de una «ley», existen excepciones. Por ejemplo, *diff*(1) (encontrar diferencias entre dos archivos) devuelve 0 para «sin diferencias», 1 para «diferencias encontradas» o 2 para un error como un argumento de nombre de archivo no válido.

[2] Los programadores de LISP encontrarán esta idea familiar.

Más específicamente, ahora podemos mejorar la función *pushd* que vimos en el último capítulo:

```
function pushd {                        # empuja el directorio actual a la pila
    dirname=$1
    cd ${dirname:?"missing directory name."}
    DIRSTACK="$dirname $DIRSTACK"
    print "$DIRSTACK"
}
```

Esta función ahora verifica si el argumento es un nombre de directorio válido antes de intentar cambiar al directorio y agregarlo a la pila. Si el nombre de directorio no es válido, imprime un mensaje de error.

Sin embargo, la función reacciona de manera engañosa cuando se proporciona un argumento que no es un directorio válido. En caso de que no lo hayas entendido al leer el último capítulo, aquí está lo que sucede: el comando *cd* falla, dejándote en el mismo directorio en el que estabas. Esto también es apropiado. Pero luego, la tercera línea de código empuja el directorio incorrecto a la pila de todos modos, y la última línea imprime un mensaje que te hace creer que la operación fue exitosa.

Necesitamos evitar que el directorio incorrecto se agregue a la pila e imprimir un mensaje de error. Aquí tienes cómo podemos hacer esto:

```
function pushd {                              # empujar el directorio actual a la pila
  dirname=$1
  if cd ${dirname:?"missing directory name."}   # si el cd tuvo exito
  then
    DIRSTACK="$dirname $DIRSTACK"
    print "$DIRSTACK"
  else
    print still in $PWD.
  fi
}
```

La llamada a *cd* está ahora dentro de una construcción `if`. Si *cd* tiene éxito, devuelve 0; las dos siguientes líneas de código se ejecutan, terminando la operación *pushd*. Pero si *cd* falla, devuelve con estado de salida 1, y *pushd* imprime un mensaje diciendo que no ha ido a ninguna parte.

Normalmente puedes confiar en que los comandos integrados y las utilidades estándar de Unix devuelvan los estados de salida apropiados, pero ¿qué pasa con tus propios scripts y funciones de shell? Por ejemplo, nos gustaría que *pushd* devolviera un estado apropiado para poder usarlo también en una sentencia `if`:

```
if pushd some-directory
then
  what we need to do
else
  handle problem case
fi
```

El problema es que el estado de salida se restablece con cada comando, por lo que «desaparece» si no se guarda inmediatamente. En esta función, el estado de salida del *cd* incorporado desaparece cuando se ejecuta la sentencia *print* (y establece su propio estado de salida).

Por lo tanto, necesitamos guardar el estado que *cd* establece y usarlo como estado de salida de toda la función. Dos características del shell que aún no hemos visto nos facilitan el camino. La primera es la variable especial del shell `?`, cuyo valor (`$?`) es el estado de salida del último comando que se ejecutó. Por ejemplo:

```
cd baddir
print $?
```

hace que el shell imprima 1, mientras que:

```
cd goodir
print $?
```

hace que el shell imprima 0.

Return

La segunda característica que necesitamos es la sentencia *return N*, que hace que el script o función que lo rodea salga con el estado de salida *N*. *N* es en realidad opcional; por defecto es el valor de salida del último comando que se ejecutó. Los scripts que terminan sin una sentencia *return* (es decir, todos los que hemos visto hasta ahora) devuelven lo que haya devuelto la última sentencia. Si utiliza *return* dentro de una función, simplemente sale de la función. (Por el contrario, la sentencia *exit N* sale de todo el script, sin importar lo profundo que esté anidado en funciones).

Volviendo a nuestro ejemplo: guardamos el estado de salida en ambas ramas del `if`, para poder utilizarlo cuando hayamos terminado:

```
function pushd {                                # push current directory onto stack
  dirname=$1
  if cd ${dirname:?"missing directory name."}  # if cd was successful
  then
    es=$?
    DIRSTACK="$dirname $DIRSTACK"
    print $DIRSTACK
  else
    es=$?
    print still in $PWD.
  fi
  return $es
}
```

La asignación `es=$?` guarda el estado de salida de *cd* en la variable es; la última línea lo devuelve como estado de salida de la función.

Los estados de salida no son muy útiles para otra cosa que no sea su propósito. En particular, puede verse tentado a utilizarlos como «valores de retorno» de funciones, como haría con funciones en C o Pascal. Esto no funcionará; en su lugar debe utilizar variables o sustitución de comandos para simular este efecto.

Ejemplo avanzado: anular el comando integrado

Utilizando el estado de salida y el comando *return*, y aprovechando el orden de búsqueda de comandos del shell, podemos escribir una función *cd* que anule el comando incorporado *command*.

Supongamos que queremos que nuestra función *cd* imprima automáticamente los directorios antiguo y nuevo. He aquí una versión para poner en su *.profile* o archivo de entorno:

```
function cd {
  command cd "$@"
  es=$?
  print "$OLDPWD -> $PWD"
  return $es
}
```

Esta función se basa en el orden de búsqueda de los comandos enumerados en el último capítulo. *cd* es un comando incorporado no especial, lo que significa que se encuentra *después* de las funciones. Por lo tanto, podemos nombrar nuestra función *cd*, y el shell la encontrará primero.

¿Pero cómo llegamos al «verdadero» comando *cd*? Lo necesitamos para cambiar de directorio. La respuesta es el comando incorporado llamado, curiosamente, *command*. Su trabajo es hacer exactamente lo que necesitamos: omitir cualquier función nombrada por el primer argumento, en su lugar encontrar el comando incorporado o externo y ejecutarlo con los argumentos suministrados. En el shell de Korn, el uso de *command* seguido de uno de los comandos incorporados especiales evita que los errores en ese comando aborten el script. (Esto resulta ser un mandato de POSIX).

ADVERTENCIA: El comando incorporado *command* no es especial. Si defines una función llamada *command*, ya no hay forma de llegar al real (excepto eliminando la función, por supuesto).

De todos modos, volvamos al ejemplo. La primera línea utiliza *command* para ejecutar *cd*. Luego guarda el estado de salida en `es`, como hicimos antes, para que pueda ser devuelto al programa que llama o al shell interactivo. Finalmente, imprime el mensaje deseado y devuelve el estado de salida guardado. Veremos una «envoltura» más sustancial para *cd* en el Capítulo 7.

Estado de salida de canalización (pipeline)

El estado de salida de un único comando es un simple número, cuyo valor, como hemos visto, está disponible en la variable especial `$?` ¿Pero qué pasa con una tubería? Después de todo, puedes conectar un número arbitrario de comandos mediante tuberías. ¿El estado de salida de una tubería es el del primer comando, el del último, o algún comando intermedio? Por defecto, es el estado de salida del *último* comando de la tubería. (Esto es requerido por POSIX).

La ventaja de este comportamiento es que está bien definido. Si un proceso falla, se sabe que fue el último comando el que falló. Pero si algún proceso intermedio en el pipeline falló, usted no lo sabe. La opción *set -o pipefail* le permite cambiar este comportamiento.[3] Al activar esta opción, el estado de salida de la canalización cambia al del último comando que falló. Si ningún comando falla, el estado de salida es 0. Esto todavía no le dice qué comando en una tubería falló, pero al menos se puede decir que algo salió mal en alguna parte y tratar de tomar medidas correctivas.

[3]Esta opción está disponible a partir de *ksh93g*.

Interpretación de los valores de estado de salida

Para *ksh93*, los valores de estado de salida para los comandos incorporados y varios casos excepcionales se han regularizado de la siguiente manera:

Tabla 5.1: Valores de estado de salida

Valor	Significado
1-125	Comando finalizado con fallo
2	Uso no válido, con mensaje de uso (comandos integrados)
126	Comando encontrado, pero el archivo no es ejecutable
127	Comando no encontrado
128-255	Comando externo finalizado con fallo
≥ 256	Comando muerto con una señal; restar 256 para obtener el número de señal

Las señales son una función más avanzada; se describen en el Capítulo 8.

5.1.2. Combinaciones de estado de salida

Una de las partes más oscuras de la sintaxis del shell de Korn le permite combinar estados de salida de forma lógica, de modo que pueda probar más de una cosa a la vez.

La sintaxis *statement1 && statement2* significa, «ejecuta statement1, y si su estado de salida es 0, ejecuta statement2». La sintaxis *statement1 || statement2* es la inversa: significa «ejecuta la sentencia1, y si su estado de salida no es 0, ejecuta la sentencia2».

A primera vista, parecen construcciones «if/then» y «if not/then», respectivamente. Pero en realidad están pensadas para su uso dentro de las condiciones de las construcciones `if`, como comprenderán fácilmente los programadores de C.

Es mucho más útil pensar en estas construcciones como «y» y «o», respectivamente. Piensa en esto:

```
if statement1 && statement2
then
  ...
fi
```

En este caso, se ejecuta la *statement1*. Si devuelve un estado 0, entonces presumiblemente se ejecutó sin error. A continuación se ejecuta la *statement2*. La cláusula `then` se ejecuta si *statement2* devuelve un estado 0. Por el contrario, si *statement1* falla (devuelve un estado de salida distinto de cero), *statement2* ni siquiera se ejecuta; la «última sentencia» de la condición era *statement1*, que falló – por lo que la cláusula `then` no se ejecuta. En conjunto,

es justo concluir que la cláusula `then` se ejecuta si tanto *statement1* como *statement2* han tenido éxito.

Del mismo modo, considere esto:

```
if statement1 || statement2
then
  ...
fi
```

Si la *statement1* tiene éxito, la *statement2* no se ejecuta. Esto convierte a la *statement1* en la última sentencia, lo que significa que la cláusula `then` se ejecuta. Por otro lado, si la *statement1* falla, se ejecuta la *statement2*, y si la cláusula `then` se ejecuta o no depende del éxito de la *statement2*. El resultado es que la cláusula `then` se ejecuta si la *statement1* o la *statement2* tienen éxito.

Como ejemplo sencillo, supongamos que necesitamos escribir un script que compruebe la presencia de dos palabras en un fichero y que simplemente imprima un mensaje diciendo si alguna de las palabras está en el fichero o no. Podemos utilizar *grep* para esto: devuelve el estado de salida 0 si encuentra la cadena dada en su entrada, distinto de cero si no:

```
filename=$1
word1=$2
word2=$3
if grep $word1 $filename > /dev/null || grep $word2 $filename > /dev/null
then
  print "$word1 or $word2 is in $filename."
fi
```

Para asegurarnos de que todo lo que obtenemos es el estado de salida, hemos redirigido la salida de ambas invocaciones de *grep* al archivo especial */dev/null*, que se conoce coloquialmente como el «cubo de bits». Cualquier salida dirigida a */dev/null* desaparece efectivamente. Sin esta redirección, la salida incluiría las líneas coincidentes que contienen las palabras, así como nuestro mensaje. (Algunas versiones de *grep* admiten una opción `-s` para «silencioso», es decir, sin salida. POSIX *grep* utiliza `-q`, que significa «silencioso». La solución más portable es redirigir la salida a */dev/null*, como hemos hecho aquí).

La cláusula `then` de este código se ejecuta si cualquiera de las dos sentencias *grep* tiene éxito. Ahora supongamos que queremos que el script diga si el fichero de entrada contiene o no ambas palabras. He aquí cómo hacerlo:

```
filename=$1
word1=$2
word2=$3 if grep $word1 $filename > /dev/null && grep $word2 $filename > /dev/null
then
```

```
  print "$word1 and $word2 are both in $filename."
fi
```

Una nota menor: cuando se usan con comandos, `&&` y `||` tienen la misma precedencia. Sin embargo, cuando se usan dentro de `[[...]]` (de lo que hablaremos en breve), `&&` tiene mayor precedencia que `||`.

Veremos más ejemplos de estos operadores lógicos más adelante en este capítulo y en el código para el depurador *kshdb* en el Capítulo 9.

5.1.3. Inversión del sentido de prueba

A veces, la forma más natural de expresar una condición es en negativo. («Si Dave no está ahí, entonces...») Supongamos que necesitamos saber que ninguna de las dos palabras está en un archivo fuente. En la mayoría de los scripts, cuando este es el caso, verá código como este:

```
if grep $word1 $filename > /dev/null || grep $word2 $filename > /dev/null
then
  : # do nothing
else
  print "$word1 and $word2 are both absent from $filename."
fi
```

El comando *:* no hace nada. El significado, entonces, es «si palabra1 o palabra2 están presentes en nombre de fichero, no hagas nada; si no, imprime un mensaje». El shell de Korn le permite hacer esto de forma más elegante utilizando la palabra clave ! (introducida en POSIX): t

```
filename=$1
word1=$2
word2=$3
if ! grep $word1 $filename > /dev/null &&
  ! grep $word2 $filename > /dev/null
then
  print "$word1 and $word2 are both absent from $filename."
fi
```

5.1.4. Pruebas de estado

Los estados de salida son las únicas cosas que una construcción `if` puede comprobar. Pero eso no significa que sólo pueda comprobar si los comandos se ejecutaron correctamente o no. El shell proporciona una forma de probar una variedad de condiciones con la construcción

`[[...]]`. [4]

Puede utilizar la construcción para comprobar muchos atributos diferentes de un archivo (si existe, qué tipo de archivo es, cuáles son sus permisos y propiedad, etc.), comparar dos archivos para ver cuál es más nuevo, hacer comparaciones y coincidencias de patrones en cadenas, y mucho más.

`[[ condición ]]` es en realidad una sentencia como cualquier otra, salvo que lo único que hace es devolver un estado de salida que indica si la condición es verdadera. Por lo tanto, encaja dentro de la sintaxis de las sentencias *if* de la construcción *if*.

Comparación de cadenas

Los corchetes dobles (`[[...]]`) rodean expresiones que incluyen varios tipos de *operadores*. Empezaremos con los operadores de comparación de cadenas, que se enumeran en la Tabla 5.2. (Observe que no hay operadores para «mayor o igual» o «menor o igual». (Observe que no hay operadores para «mayor que o igual» o «menor que o igual»). En la tabla, *str* se refiere a una expresión con un valor de cadena, y *pat* se refiere a un patrón que puede contener comodines (igual que los patrones en los operadores de manejo de cadenas que vimos en el capítulo anterior). Tenga en cuenta que estos operadores comparan los valores lexicográficos de las cadenas, por lo que “10” < “2”.

Tabla 5.2: Operadores de comparación de cadenas

Operador	**Verdadero si...**
str	*str* no es nulo
str == pat	*str* coincide con *pat*
str = pat	*str* coincide con *pat* (obsoleto)
str != pat	*str* no coincide con *pat*
str1 < str2	*str1* es menor que *str2*
str1 > str2	*str1* es mayor que *str2*
-n str	*str* no es nulo (tiene una longitud mayor que 0)
-z str	*str* es nulo (tiene longitud 0)

[4] El shell de Korn también acepta los comandos `[...]` y *test*. (Hay comandos incorporados en todas las versiones de *ksh*; se comportan como las versiones externas originales). La construcción `[[...]]` tiene muchas más opciones y está mejor integrada en el lenguaje del shell de Korn: específicamente, la división de palabras y la expansión de comodines no se hacen dentro de `[[` y `]]`, haciendo las citas menos necesarias. Además, siempre puede distinguir los operadores de los operandos, ya que los operadores no pueden ser el resultado de la expansión.

Podemos utilizar uno de estos operadores para mejorar nuestra función *popd*, que reacciona mal si se intenta hacer pop y la pila está vacía. Recordemos que el código para *popd* es:

```
function popd {                  # cd to top, pop it off stack
  top=${DIRSTACK%% *}
  DIRSTACK=${DIRSTACK#* }
  cd $top
  print "$PWD"
}
```

Si la pila está vacía, `$DIRSTACK` es la cadena nula, al igual que la expresión `$DIRSTACK%% *`. Esto significa que cambiará a su directorio personal; en su lugar, queremos que *popd* imprima un mensaje de error y no haga nada.

Para ello, debemos comprobar si la pila está vacía, es decir, si `$DIRSTACK` es nulo o no. He aquí una forma de hacerlo:

```
function popd {                  # pop directory off the stack, cd there
  if [[ -n $DIRSTACK ]];
    then top=${DIRSTACK%% *}
    DIRSTACK=${DIRSTACK#* }
    cd $top
    print "$PWD"
  else
    print "stack empty, still in $PWD."
    return 1
  fi
}
```

Observe que en lugar de poner `then` en una línea separada, lo ponemos en la misma línea que el `if` después de un punto y coma, que es el carácter separador de sentencias estándar del shell. (Hay una sutileza aquí. El shell sólo reconoce palabras clave como `if` y `then` cuando están al principio de una sentencia. Esto es para que pueda escribir, por ejemplo, `print if then else is neat` sin obtener errores de sintaxis. Las nuevas líneas y el punto y coma separan las sentencias. Así, el `then` en la misma línea que el `if` es reconocido correctamente después de un punto y coma, mientras que sin el punto y coma, no lo sería).

Podríamos haber utilizado operadores distintos de *-n*. Por ejemplo, podríamos haber utilizado *-z* e intercambiado el código en las cláusulas `then` y `else`. También podríamos haber utilizado:

```
if [[ $DIRSTACK == "" ]]; then
  ...
```

[[...]] vs el comando *test* y [...]

Escribimos nuestra prueba `[[ $DIRSTACK == '''' ]]`. Este no es el uso correcto para la sintaxis más antigua `[...]` o *test*.

En esta sintaxis, que el shell de Korn todavía soporta, y que es todo lo que tiene en el shell de Bourne, si `$DIRSTACK` se evalúa como una cadena nula, el shell se quejará de que falta un argumento. Esto lleva al requisito de encerrar ambas cadenas entre comillas dobles (`[ ''$DIRSTACK>> = '''' ]`), que es la forma más legible de hacerlo, o al truco común de añadir un carácter extra delante de las cadenas, como así: `[ x$DIRSTACK = x ]`. Esto último funciona, ya que si `$DIRSTACK` es nulo, el comando `[...]` sólo ve los dos caracteres `x`, pero no es muy obvio lo que está pasando, especialmente para el novato.

Tenga en cuenta también que el operador preferido del shell de Korn es `==`, mientras que test requiere un único carácter `=`.

Mientras limpiamos el código que escribimos en el último capítulo, corrijamos el manejo de errores en el script principal (Tarea 4-1). El código de ese script es:

```
filename=${1:?"filename missing."}
howmany=${2:-10}
sort -nr $filename | head -$howmany
```

Recuerda que si omites el primer argumento (el nombre de archivo), el shell imprime el mensaje `highest: 1: filename missing.` Podemos mejorarlo sustituyendo un mensaje de «uso» más estándar:

```
if [[ -z $1 ]]; then
  print 'usage: highest filename [N]'
else
  filename=$1
  howmany=${2:-10}
  sort -nr $filename | head -$howmany
fi
```

Se considera un estilo de programación mejor encerrar todo el código en el `if-then-else`, pero ese código puede volverse confuso si estás escribiendo un script largo en el que necesitas verificar errores y salir en varios puntos del camino. Por lo tanto, un estilo más común para la programación en shell es el siguiente:

```
if [[ -z $1 ]]; then
  print 'usage: highest filename [-N]'
  exit 1
fi
filename=$1
```

```
howmany=${2:-10}
sort -nr $filename | head -$howmany
```

La instrucción `exit` informa a cualquier programa llamante que necesita saber si se ejecutó correctamente o no. (También puedes usar `return`, pero consideramos que `return` debería reservarse para su uso en funciones).

Como ejemplo de los operadores `==` y `!=`, podemos agregar a nuestra solución para la Tarea 4-2, el script principal de un compilador de C. Recuerda que se nos proporciona un nombre de archivo que termina en `.c` (el archivo de código fuente), y necesitamos construir un nombre de archivo que sea el mismo pero que termine en `.o` (el archivo de código objeto). Las modificaciones que haremos tienen que ver con otros tipos de archivos que se pueden pasar a un compilador de C.

Acerca de los compiladores C

Antes de llegar al código shell, es necesario entender algunas cosas sobre los compiladores de C. Ya sabemos que traducen el código fuente de C a código objeto. En realidad, forman parte de sistemas de compilación que también realizan otras tareas. El término «compilador» se utiliza a menudo en lugar de «sistema de compilación», así que lo usaremos en ambos sentidos.

Aquí nos interesan dos tareas que realizan los compiladores aparte de compilar código C: pueden traducir código de lenguaje ensamblador a código objeto y pueden enlazar archivos de código objeto para formar un programa ejecutable.

El lenguaje ensamblador trabaja a un nivel muy cercano al del ordenador: cada sentencia en ensamblador puede traducirse directamente en una sentencia de código objeto, a diferencia de C u otros lenguajes de alto nivel, en los que una sola sentencia fuente puede traducirse en docenas de instrucciones de código objeto. Traducir un archivo de código en lenguaje ensamblador a código objeto se denomina, como es lógico, ensamblar el código.

Aunque mucha gente considera que el lenguaje ensamblador es algo pintorescamente anticuado -como una máquina de escribir en la era del tratamiento de textos WYSIWYG y la autoedición-, algunos programadores siguen necesitándolo cuando tratan detalles precisos del hardware informático. No es raro que un programa conste de varios archivos de código en un lenguaje de alto nivel (como C o C++) y algunas rutinas de bajo nivel en lenguaje ensamblador.

La otra tarea de la que nos ocuparemos se llama enlazado. La mayoría de los programas del mundo real, a diferencia de los asignados a una clase de programación de primer curso, constan de varios archivos de código fuente, posiblemente escritos por varios programadores diferentes. Estos archivos se compilan en código objeto; después, el código objeto debe combinarse para formar el programa final ejecutable. La tarea de combinar suele denominarse «enlazar»: cada componente del código objeto suele contener referencias a otros componentes, y estas referencias deben resolverse o «enlazarse» entre sí.

Los sistemas de compilación C son capaces de ensamblar archivos de lenguaje ensamblador en código objeto y enlazar archivos de código objeto en ejecutables. En concreto, un compilador llama a un ensamblador independiente para que se ocupe del código ensamblador y a un enlazador (también conocido como «cargador», «cargador de enlace» o «editor de enlace») para que se ocupe de los archivos de código objeto. Estas herramientas separadas se conocen en el mundo Unix como *as* y *ld*, respectivamente. El compilador de C propiamente dicho se invoca con el comando *cc*.

Podemos expresar todos estos pasos en términos de los sufijos de los archivos pasados como argumentos al compilador de C. Básicamente, el compilador hace lo siguiente:

1. Si el argumento termina en `.c` es un archivo fuente C; compilar en un archivo de código objeto `.o`.
2. Si el argumento termina en `.s`, es lenguaje ensamblador; ensamble en un archivo `.o`.
3. Si el argumento termina en `.o`, no haga nada; guárdelo para el paso de vinculación posterior.
4. Si el argumento termina en algún otro sufijo, imprime un mensaje de error y sale.[5]
5. Enlaza todos los archivos de código objeto `.o` en un archivo ejecutable llamado `a.out`. Este archivo suele renombrarse a algo más descriptivo.

El paso 3 permite reutilizar archivos de código objeto ya compilados (o ensamblados) para crear otros ejecutables. Por ejemplo, un archivo de código objeto que implemente una interfaz para una unidad de CD-ROM podría ser útil en cualquier programa que lea CD-ROMs.

La Figura 5.1 debería hacer más claro el proceso de compilación; muestra cómo el compilador procesa los archivos fuente C `a.c` y `b.c`, el archivo de lenguaje ensamblador `c.s`, y el

[5] A efectos de este ejemplo. Sabemos que esto no es estrictamente cierto en la vida real.

archivo de código objeto ya compilado `d.o`. En otras palabras, muestra cómo el compilador maneja el comando `cc a.c b.c c.s d.o`.

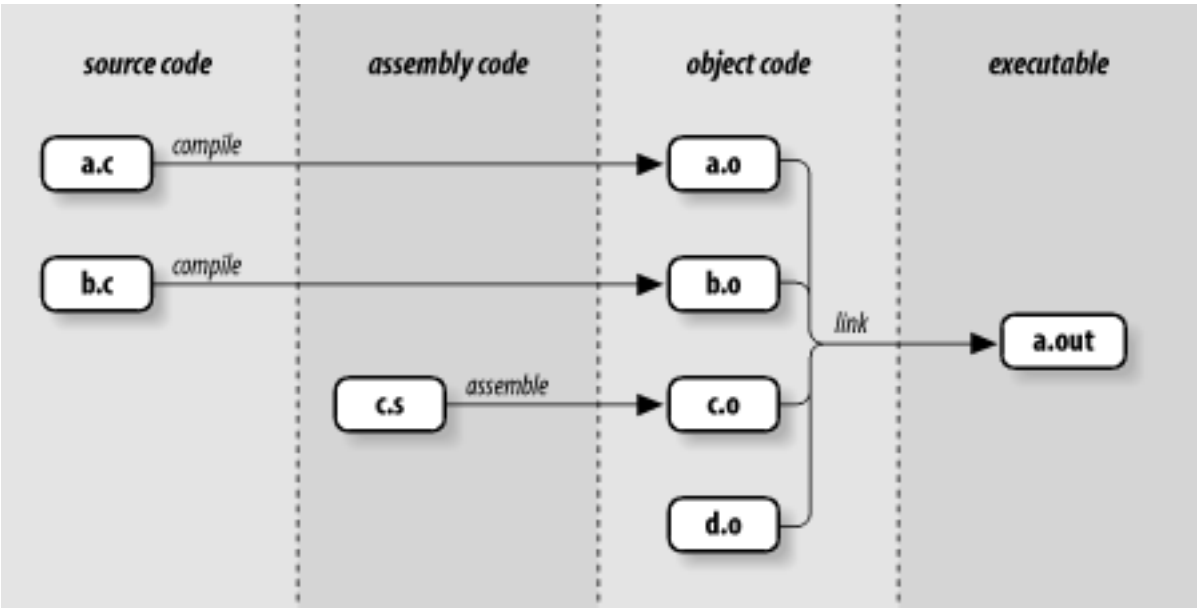

Figura 5.1: Archivos producidos por un compilador C

He aquí cómo empezaríamos a implementar este comportamiento en un script de shell. Supongamos que la variable `filename` contiene el argumento en cuestión, y que *ccom* es el nombre del programa que realmente compila un fichero fuente C en código objeto. Supongamos además que *ccom* y as (ensamblador) toman como argumentos los nombres de los ficheros fuente y objeto:

```
if [[ $filename == *.c ]]; then
  objname=${filename%.c}.o
  ccom "$filename" "$objname"
elif [[ $filename == *.s ]]; then
  objname=${filename%.s}.o
  as "$filename" "$objname"
elif [[ $filename != *.o ]]; then
  print "error: $filename is not a source or object file."
  exit 1
fi
further processing ...
```

Recuerde del capítulo anterior que la expresión `$filename%.c.o` elimina `.c` del nombre de archivo y añade `.o`; `$filename%.s.o` hace lo mismo con los archivos que terminan en `.s`.

El «procesamiento posterior» es el paso de enlace, que veremos cuando completemos este ejemplo más adelante en el capítulo.

Comprobación de atributos de archivo

El otro tipo de operador que puede utilizarse en expresiones condicionales comprueba si un fichero tiene determinadas propiedades. Existen 24 operadores de este tipo. Aquí cubrimos los de interés más general; el resto se refieren a arcanos como sticky bits, sockets y descriptores de fichero, y por tanto son de interés sólo para programadores de sistemas. Consulte el Apéndice B para ver la lista completa. La Tabla 5.3 enumera las que nos interesan ahora.

Tabla 5.3: Operadores de atributos de archivos

Operador	Verdadero si...
`-e` *file*	*file* existe
`-d` *file*	*file* es un directorio
`-f` *file*	*file* es un archivo regular (por ejemplo, no un directorio u otro tipo especial de archivo)
`-L` *file*	*file* es un enlace simbólico
`-r` *file*	Tiene permisos de lectura en *file*
`-s` *file*	*file* existe y no está vacío
`-w` *file*	Tiene permisos de escritura en *file*
`-x` *file*	Tienes permiso de ejecución sobre *file* o permiso de búsqueda de directorio si es un directorio
`-O` *file*	Su propio archivo (el UID efectivo coincide con el del archivo)
`-G` *file*	Su ID de grupo efectivo es el mismo que el del fichero
file1 `-nt` *file2*	*file1* es más nuevo que *file2*[6]
file1 `-ot` *file2*	*file1* es más antiguo que *file2*
file1 `-ef` *file2*	*file1* y *file2* son el mismo archivo

Antes de llegar a un ejemplo, debes saber que las expresiones condicionales dentro de [[y]] también pueden combinarse usando los operadores lógicos && y ||, tal como vimos con los comandos de shell simples en la Sección 5.1.2, anteriormente en este capítulo. También es posible combinar comandos del shell con expresiones condicionales utilizando operadores lógicos, como en este caso:

```
if command && [[ condition ]]; then
  ...
```

El Capítulo 7 contiene un ejemplo de esta combinación.

También puede negar el valor verdadero de una expresión condicional precediéndola de un signo de exclamación (!), de modo que `! expr` se evalúe como verdadero sólo si `expr` es falso. Además, puede hacer expresiones lógicas complejas de operadores condicionales agrupándolos con paréntesis. (Resulta que esto también es cierto fuera de la construcción

[6] En concreto, los operadores `-nt` y `-ot` comparan los *tiempos de modificación* de dos archivos.

[[...]]). Como veremos en el Capítulo 8, la construcción *(statement list)* ejecuta la lista de sentencias en un subshell, cuyo estado de salida es el de la última sentencia de la lista).

Así es como usaríamos dos de los operadores de fichero para embellecer (una vez más) nuestra función *pushd*. En lugar de hacer que *cd* determine si el argumento dado es un directorio válido – es decir, devolviendo con un estado de salida malo si no lo es – podemos hacer la comprobación nosotros mismos. Aquí está el código:

```
function pushd {                # empujar el directorio actual a la pila
  dirname=$1
  if [[ -d $dirname && -x $dirname ]]; then
    cd "$dirname"
    DIRSTACK="$dirname DIRSTACK"
    print "$DIRSTACK"
  else
    print "still in $PWD."
    return 1
  fi
}
```

La expresión condicional evalúa a verdadero sólo si el argumento `$1` es un directorio (*-d*) y el usuario tiene permiso para cambiar a él (*-x*).[7] Note que esta condicional también maneja el caso donde el argumento falta: `$dirname` es nulo, y como la cadena nula no es un nombre de directorio válido, la condicional fallará.

La Tarea 5-1 presenta un ejemplo más completo del uso de los operadores de archivo.

Tarea 5-1

Escribe un script que imprima esencialmente la misma información que `ls -l` pero de una forma más fácil de usar.

Aunque esta tarea requiere un código relativamente prolijo, es una aplicación directa de muchos de los operadores de ficheros:

```
if [[ ! -e $1 ]]; then
  print "file $1 does not exist."
  return 1
fi
fi [[ [[ -d $1 ]]; then
  print -n "$1 is a directory that you may "
  if [[ ! -x $1 ]]; then
    print -n "not "
  fi
  print "search."
```

[7] Recuerde que el mismo indicador de permiso que determina el permiso de ejecución en un archivo normal determina el permiso de búsqueda en un directorio. Por eso el operador -x comprueba ambas cosas dependiendo del tipo de fichero.

```
elif [[ -f $1 ]]; then
  print "$1 is a regular file."
else
  print "$1 is a special type of file."
fi
if [[ -O $1 ]]; then
  print 'you own the file.'
else
  print 'you do not own the file.'
fi
if [[ -r $1 ]]; then
  print 'you have read permission on the file.'
fi
if [[ -w $1 ]]; then
  print 'you have write permission on the file.'
fi
if [[ -x $1 && ! -d $1 ]]; then
  print 'you have execute permission on the file.'
fi
```

Llamaremos a este script *fileinfo*. Así es como funciona:

- La primera condicional comprueba si el archivo dado como argumento no existe (el signo de exclamación es el operador «not»; los espacios que lo rodean son obligatorios). Si el fichero no existe, el script imprime un mensaje de error y sale con el estado de error.
- La segunda condicional comprueba si el fichero es un directorio. Si es así, la primera imprime parte de un mensaje; recuerde que la opción *-n* indica a *print* que no imprima una nueva línea al final. La condicional interna comprueba si no tienes permiso de búsqueda en el directorio. Si no tiene permiso de búsqueda, se añade la palabra «not» al mensaje parcial. A continuación, el mensaje se completa con «search» y una nueva línea.
- La cláusula `elif` comprueba si el fichero es un fichero normal; en caso afirmativo, imprime un mensaje.
- La cláusula *else* tiene en cuenta los distintos tipos de ficheros especiales de los sistemas Unix recientes, como sockets, dispositivos, ficheros FIFO, etc. Asumimos que el usuario ocasional no está interesado en sus detalles.
- La siguiente condicional comprueba si eres el propietario del fichero (es decir, si su ID de propietario es el mismo que tu ID de usuario efectivo). Si es así, imprime un mensaje diciendo que usted es el propietario. (Los ID reales y efectivos de Usuario y

Grupo se explican en el Capítulo 10.)

- Las dos condiciones siguientes comprueban sus permisos de lectura y escritura sobre el fichero.
- La última condicional comprueba si puedes ejecutar el fichero. Comprueba si tiene permiso de ejecución y si el fichero no es un directorio. (Si el archivo fuera un directorio, el permiso de ejecución significaría realmente el permiso de búsqueda de directorio).

Como ejemplo de la salida de *fileinfo*, suponga que hace un `ls -l` de su directorio actual y contiene estas líneas:

```
-rwxr-xr-x   1   billr   other    594     May   28   09:49   bob
-rw-r-r-     1   billr   other    42715   Apr   21   23:39   custom.tbl
drwxr-xr-x   2   billr   other    64      Jan   12   13:42   exp
-r-r-r-      1   root    other    557     Mar   28   12:41   lpst
```

custom.tbl y *lpst* son archivos de texto normales, *exp* es un directorio y *bob* es un script de shell. Escribir `fileinfo bob` produce esta salida:

```
bob is a regular file.
you own the file.
you have read permission on the file.
you have write permission on the file.
you have execute permission on the file.
```

Al escribir *fileinfo* `custom.tbl` se obtiene lo siguiente:

```
custom.tbl is a regular file.
you own the file.
you have read permission on the file.
you have write permission on the file.
```

Escribiendo `fileinfo exp` se obtiene esto:

```
exp is a directory that you may search.
you own the file.
you have read permission on the file.
you have write permission on the file.
```

Por último, al escribir `fileinfo lpst` se obtiene lo siguiente:

```
lpst is a regular file.
you do not own the file.
you have read permission on the file.
```

Condicionales aritmétricos

El shell también proporciona un conjunto de pruebas *aritméticas*. Éstas son diferentes de las comparaciones de *cadenas de caracteres* como < y >, que comparan valores *lexicográficos* de cadenas, no valores numéricos. Por ejemplo, «6» es mayor que «57» lexicográficamente, al igual que «p» es mayor que «ox», pero, por supuesto, ocurre lo contrario cuando se comparan como números.

En la Tabla 5.4 se resumen los operadores de comparación aritmética. Los programadores de Fortran encontrarán su sintaxis ligeramente familiar.

Tabla 5.4: Operadores aritméticos de prueba

Prueba	Comparación	Prueba	Comparación
`-lt`	Menor que	`-gt`	Mayor que
`-le`	Menor o igual que	`-ge`	Mayor o igual que
`-eq`	Igual	`-ne`	No igual

Le resultarán muy útiles en el contexto de las variables numéricas que veremos en el próximo capítulo. Son necesarias si desea combinar pruebas numéricas con otros tipos de pruebas dentro de la misma expresión condicional.

Sin embargo, el shell tiene una sintaxis separada para expresiones condicionales que involucran sólo números. (Esta sintaxis se trata en el Capítulo 6.) Es considerablemente más eficiente, así como más general, por lo que debería utilizarla con preferencia a los operadores de prueba aritmética enumerados anteriormente.

De hecho, parte de la documentación de *ksh93* considera obsoletas estas condicionales numéricas. Por lo tanto, si necesita combinar [[...]] y pruebas numéricas, hágalo utilizando los operadores `!`, `&&` y `||` del intérprete de comandos fuera de [[...]], en lugar de dentro de ellos. De nuevo, cubriremos los condicionales numéricos del shell en el próximo capítulo.

5.2. for

La mejora más obvia que podríamos hacer al script anterior es la capacidad de informar sobre varios archivos en lugar de sólo uno. Pruebas como *-e* y *-d* sólo toman argumentos individuales, por lo que necesitamos una manera de llamar al código una vez para cada archivo dado en la línea de comandos.

La forma de hacer esto – de hecho, la forma de hacer muchas cosas con el shell de Korn – es con una construcción de bucle. La más simple y ampliamente aplicable de las construcciones

de bucle del shell es el bucle `for`. Usaremos `for` para mejorar *fileinfo* pronto.

El bucle for le permite repetir una sección de código un número fijo de veces. Durante cada vez que se repite el código (conocido como iteración), una variable especial llamada *variable de bucle* se establece a un valor diferente; de esta forma cada iteración puede hacer algo ligeramente diferente.

El bucle *for* es algo, pero no totalmente, similar a sus homólogos en lenguajes convencionales como C y Pascal. La principal diferencia es que el bucle *for* del shell no le permite especificar un número de veces para iterar o un rango de valores sobre los que iterar; en su lugar, sólo le permite dar una lista fija de valores. En otras palabras, con el bucle *for* normal, no puedes hacer nada como este código tipo Pascal, que ejecuta las sentencias 10 veces:

```
for x := 1 to 10 do
begin
  statements ...
end
```

(Para ello necesitas el bucle aritmético *for*, que veremos en el Capítulo 6).

Sin embargo, el bucle *for* es ideal para trabajar con argumentos en la línea de órdenes y con conjuntos de ficheros (por ejemplo, todos los ficheros de un directorio determinado). Veremos un ejemplo de cada uno de ellos. Pero primero, aquí está la sintaxis para la construcción *for*:

```
for name [in list]
do
  statements that can use $name ...
done
```

list es una lista de nombres. (Si se omite en *list*, la lista por defecto es "$@", es decir, la lista entrecomillada de argumentos de la línea de comandos, pero siempre proporcionamos en *list* en aras de la claridad). En nuestras soluciones a la siguiente tarea, mostramos dos formas sencillas de especificar listas.

En *ksh93* hay una interesante interacción entre el bucle *for* y las variables *nameref* (ver Capítulo 4). Si la variable de control es un *nameref*, entonces cada elemento de la lista de nombres puede ser una variable shell diferente, y el shell asigna el *nameref* a cada variable sucesivamente. Por ejemplo:

```
$ first="I am first"                          # Initialize test variables
$ second="I am in the middle"
$ third="I am last"
$ nameref refvar=first                        # Create nameref
$ for refvar in first second third ; do       # Loop over variables
```

```
> print "refvar -> ${!refvar}, value: $refvar"  # Print referenced var, value
> done
refvar -> first, value: I am first
refvar -> second, value: I am in the middle
refvar -> third, value: I am last
$ print ${!refvar}, $refvar                     # Show final state
third, I am last
```

El bucle *for* es fundamental para resolver la Tarea 5-2.

Tarea 5-2

Usted trabaja en un entorno con varios ordenadores en una red local. Escribe un script de shell que te diga quién ha iniciado sesión en cada máquina de la red.

El comando *finger(1)* se puede utilizar (entre otras cosas) para encontrar los nombres de los usuarios que han iniciado sesión en un sistema remoto; el comando `finger @systemname` hace esto. Su salida depende de la versión de Unix, pero se parece a esto:

```
[motet.early.com]
Trying 127.146.63.17...
-User-  -Full name-           -What-    Idle TTY    -Console Location-
hildy   Hildegard von Bingen    ksh     2d5h p1     jem.cal (Telnet)
mikes   Michael Schultheiss     csh     1:21 r4     ncd2.cal (X display 0)
orlando Orlando di Lasso        csh     28 r7       maccala (Telnet)
marin   Marin Marais            mush    1:02 pb     mussell.cal (Telnet)
johnd   John Dowland            tcsh    17 p0       nugget.west.nobis. (X Window)
```

En esta salida, *motet.early.com* es el nombre de red completo de la máquina remota.

Suponga que los sistemas de su red se llaman *fred, bob, dave* y *pete*. Entonces el siguiente código haría el truco:

```
for sys in fred bob dave pete
do
  finger @$sys
  print
done
```

Esto funciona independientemente del sistema en el que esté conectado. Imprime una salida para cada máquina similar a la anterior, con líneas en blanco entre ellas.

Una solución ligeramente mejor sería almacenar los nombres de los sistemas en una variable de entorno. De esta forma, si se añaden sistemas a su red y necesita una lista de sus nombres en más de un script, sólo tendrá que cambiarlos en un lugar. Si el valor de una variable son varias palabras separadas por espacios (o TABS), *for* lo tratará como una lista de palabras.

Aquí está la solución mejorada. Primero, ponga líneas en su archivo *.profile* o de entorno que definan la variable `SYSNAMES` y conviértala en una variable de entorno:

```
SYSNAMES="fred bob dave pete"
export SYSNAMES
```

Entonces, el script puede tener este aspecto:

```
for sys in $SYSNAMES
do
  finger @$sys
  print
done
```

Lo anterior ilustra un uso simple de *for*, pero es mucho más común usar *for* para iterar a través de una lista de argumentos de línea de comandos. Para mostrar esto, podemos mejorar el script *fileinfo* anterior para que acepte múltiples argumentos. Primero, escribimos un poco de código «envolvente» que hace la iteración:

```
for filename in "$@" ; do
  finfo $filename
  print
done
```

A continuación, convertimos el script original en una función llamada *finfo*[8]:

```
function finfo {
  if [[ ! -e $1 ]]; then
    print "file $1 does not exist."
    return 1
  fi
  ...
}
```

El script completo consiste en el código del bucle *for* y la función anterior. Debido a que la función debe ser definida antes de que pueda ser usada, la definición de la función debe ir primero, o bien debe estar en un directorio listado tanto en `PATH` como en `FPATH`.

El script *fileinfo* funciona de la siguiente manera: en la sentencia *for*, «$@» es una lista de todos los parámetros posicionales. Para cada argumento, el cuerpo del bucle se ejecuta con `filename` fijado a ese argumento. En otras palabras, la función *fileinfo* es llamada una vez por cada valor de `$filename` como su primer argumento ($1). La llamada a *print* después de la llamada a *fileinfo* simplemente imprime una línea en blanco entre los conjuntos de información sobre cada fichero.

[8]Una función puede tener el mismo nombre que un script; sin embargo, esto no es una buena práctica de programación.

Dado un directorio con los mismos ficheros que en el ejemplo anterior, al teclear `fileinfo *` se obtendría la siguiente salida:

```
bob is a regular file. you own the file.
you have read permission on the file.
you have write permission on the file.
you have execute permission on the file.

custom.tbl is a regular file.
you own the file.
you have read permission on the file.
you have write permission on the file.

exp is a directory that you may search.
you own the file.
you have read permission on the file.
you have write permission on the file.

lpst is a regular file.
you do not own the file.
you have read permission on the file.
```

La Tarea 5-3 es una tarea de programación que explota el otro uso principal de *for*.

Tarea 5-3

Su sistema Unix tiene la capacidad de transferir archivos desde un sistema MS-DOS, pero deja intactos los nombres de archivo MS-DOS. Escribe una secuencia de comandos que traduzca los nombres de archivo de un directorio determinado del formato MS-DOS a un formato más compatible con Unix.

Los nombres de archivo en el antiguo sistema MS-DOS de Microsoft tienen el formato *FILENAME.EXT*. *FILENAME* puede tener hasta ocho caracteres; *EXT* es una extensión que puede tener hasta tres caracteres. Las letras son todas mayúsculas. Queremos hacer lo siguiente:

1. Traducir letras de mayúsculas a minúsculas.
2. Si la extensión es nula, elimine el punto.

La primera herramienta que necesitaremos para este trabajo es la utilidad Unix *tr(1)*, que traduce caracteres de uno en uno [9]. Dados los argumentos *charset1* y *charset2*, traduce los caracteres de la entrada estándar que son miembros de *charset1* a los caracteres correspondientes de *charset2*. Los dos conjuntos son rangos de caracteres encerrados entre corchetes

[9] Como veremos en el Capítulo 6, es posible hacer la traducción de mayúsculas y minúsculas dentro del shell, sin usar un programa externo. Sin embargo, ignoraremos ese hecho por ahora.

(`[...]` en forma de expresión regular estándar a la manera de *grep, awk, ed,* etc.). Más concretamente, `tr [A-Z] [a-z]` toma su entrada estándar, convierte las mayúsculas en minúsculas y escribe el texto convertido en la salida estándar [10].

Esto se encarga del primer paso en el proceso de traducción. Podemos usar un operador de cadena del shell de Korn para manejar el segundo. Aquí está el código para un script que llamaremos *dosmv*:

```
for filename in ${1:+$1/}* ; do
  newfilename=$(print $filename | tr '[A-Z]' '[a-z]')
  newfilename=${newfilename%.}
  print "$filename -> $newfilename"
  mv $filename $newfilename
done
```

El * en la construcción *for no* es lo mismo que `$*`. Es un comodín, es decir, todos los archivos de un directorio.

Este script acepta un nombre de directorio como argumento, siendo por defecto el directorio actual. La expresión `${1:+$1/}` se evalúa como el argumento ($1) con una barra añadida si se proporciona el argumento, o la cadena nula si no se proporciona. Por lo tanto, la expresión `${1:+$1/}*` completa equivale a todos los archivos del directorio indicado, o a todos los archivos del directorio actual si no se proporciona ningún argumento.

Por lo tanto, `filename` toma el valor de cada *filename* de la lista. `filename` se convierte en `newfilename` en dos pasos. (Podríamos haberlo hecho en uno, pero la legibilidad se habría resentido.) El primer paso utiliza *tr* en un canal dentro de una construcción de sustitución de comandos. Nuestro viejo amigo *print* convierte el valor de `filename` en la entrada estándar de *tr*. La salida de *tr* se convierte en el valor de la expresión de sustitución de comandos, que se asigna a `newfilename`. Así, si `$filename` fuera `DOSFILE.TXT`, `filename` se convertiría en `dosfile.txt`.

El segundo paso utiliza uno de los operadores de coincidencia de patrones del intérprete de comandos, el que elimina la coincidencia más corta que encuentra al final de la cadena. El patrón aquí es `.`, que significa un punto al final de la cadena [11]. Esto significa que la expresión `${newfilename%.}` eliminará un punto de `$newfilename` sólo si está al final de la cadena; de lo contrario, la expresión dejará `$newfilename` intacto. Por ejemplo, si

[10] Los sistemas modernos compatibles con POSIX soportan locales, que son formas de utilizar conjuntos de caracteres no ASCII de forma portable. En un sistema así, la invocación correcta de *tr* es `tr [:upper:] [:lower:]`. Sin embargo, la mayoría de los usuarios veteranos de Unix tienden a olvidar esto.

[11] Los expertos en expresiones regulares de Unix deben recordar que se trata de la sintaxis de comodines del shell, en la que los puntos no son operadores y, por lo tanto, no es necesario ocultarlos.

`$newfilename` es `dosfile.txt`, no se modificará, pero si es `dosfile.`, la expresión lo cambiará a `dosfile` sin el punto final. En cualquier caso, el nuevo valor se asigna de nuevo a `newfilename`.

La última sentencia en el cuerpo del bucle *for* realiza el renombrado de ficheros con el comando estándar de Unix *mv(1)*. Antes de eso, un comando *print* simplemente informa al usuario de lo que está ocurriendo.

Hay un pequeño problema con esta solución: si hay ficheros en el directorio dado que no son ficheros MS-DOS (en particular, si hay ficheros cuyos nombres no contienen letras mayúsculas o no contienen un punto), entonces la conversión no hará nada con esos nombres de fichero y *mv* será llamado con dos argumentos idénticos. *mv* se quejará con el mensaje: `mv: filename and filename are identical`. La solución es muy sencilla: compruebe si los nombres de archivo son idénticos:

```
for filename in ${1:+$1/}* ; do
  newfilename=$(print $filename | tr '[A-Z]' '[a-z]')
  newfilename=${newfilename%.}
  # subtlety: quote value of $newfilename to do string comparison,
  # not regular expression match
  if [[ $filename != "$newfilename" ]]; then
    print "$filename -> $newfilename"
    mv $filename $newfilename
  fi
done
```

Si está familiarizado con un sistema operativo distinto de MS-DOS y Unix, puede poner a prueba su destreza en la escritura de scripts en este punto escribiendo un script que traduzca los nombres de archivo del formato de ese sistema al formato Unix. Utilice el script anterior como guía.

En concreto, si conoce el sistema operativo OpenVMS de Compaq (nee DEC), aquí tiene un reto de programación:

1. Escriba un script llamado *vmsmv* que sea similar a dosmv pero que funcione con nombres de archivo OpenVMS en lugar de con nombres de archivo MS-DOS. Recuerde que los nombres de archivo OpenVMS terminan con punto y coma y números de versión.

2. Modifica tu script para que si hay varias versiones del mismo fichero, renombre sólo la última versión (con el número de versión más alto).

3. Modifíquelo aún más para que su script borre las versiones antiguas de los archivos.

La primera de ellas es una modificación relativamente sencilla de dosmv. La número 2 es difícil; aquí tienes una pista de estrategia:

- Desarrolle una expresión regular que coincida con los nombres de archivo OpenVMS (de todos modos, la necesitará para el número 1).
- Obtenga una lista de los nombres base (sin los números de versión) de los archivos del directorio indicado pasando *ls* por *grep* (con la expresión regular anterior), *cut* y *sort -u*. Utilice *cut* con punto y coma como «separador de campos». Utilice cut con un punto y coma como "separador de campos"; asegúrese de entrecomillar el punto y coma para que el shell no lo trate como un separador de sentencias. *sort -u* elimina los duplicados tras la ordenación. Utilice la sustitución de comandos para guardar la lista resultante en una variable.
- Utiliza un bucle *for* en la lista de nombres base. Para cada nombre, obtenga el número de versión más alto del archivo (sólo el número, no el nombre completo). Haga esto con otra tubería: pipe *ls* a través de *cut*, *sort -n*, y *tail -1*. *sort -n* ordena en orden numérico (no lexicográfico); *tail -N* muestra las últimas *N* líneas de su entrada. De nuevo, utilice la sustitución de comandos para capturar la salida de este proceso en una variable.
- Añada el número de versión más alto al nombre base; éste es el archivo que hay que renombrar en formato Unix.

Una vez que hayas completado el número 2, puedes hacer el número 3 añadiendo una sola línea de código a tu script; a ver si averiguas cómo.

Finalmente, *ksh93* proporciona el bucle *for* aritmético, que es mucho más cercano en sintaxis y estilo al bucle *for* de C. Lo presentamos en el próximo capítulo, después de discutir las capacidades aritméticas generales del shell. Lo presentamos en el próximo capítulo, después de discutir las capacidades aritméticas generales del shell.

5.3. case

La siguiente construcción de control de flujo a cubrir es `case`. Mientras que la sentencia `case` en Pascal y la sentencia `witch` similar en C pueden usarse para probar valores simples como enteros y caracteres, la construcción `case` del shell de Korn le permite probar cadenas contra patrones que pueden contener caracteres comodín. Al igual que sus equivalentes en

lenguajes convencionales, `case` permite expresar una serie de sentencias del tipo `if-then-else` de forma concisa.

La sintaxis de `case` es la siguiente

```
case expression in
  pattern1 )
    statements ;;
  pattern2 )
    statements ;;
  ...
esac
```

Cualquiera de los *patrones* puede ser en realidad varios patrones separados por caracteres «barra OR» (|, que es lo mismo que el símbolo de la tubería, pero en este contexto significa «o»). Si la *expresión* coincide con uno de los patrones, se ejecutan las sentencias correspondientes. Si hay varios patrones separados por barras OR, la expresión puede coincidir con cualquiera de ellos para que se ejecuten las sentencias asociadas. Los patrones se comprueban en orden hasta que se encuentra una coincidencia; si no se encuentra ninguna, no ocurre nada.

Esta sintaxis bastante desgarbada debería quedar más clara con un ejemplo. Una opción obvia es revisar nuestra solución a la Tarea 4-2, el front-end para el compilador C. Anteriormente en este capítulo, escribimos algo de código que procesaba archivos de entrada según sus sufijos (.c, .s, o .o para C, ensamblador, o código objeto, respectivamente).

Podemos mejorar esta solución de dos maneras. En primer lugar, podemos utilizar *for* para permitir que se procesen varios archivos a la vez; en segundo lugar, podemos utilizar case para agilizar el código:

```
for filename in "$@"; do
  case $filename in
    *.c )
      objname=${filename%.c}.o
      ccom "$filename" "$objname" ;;
    *.s )
      objname=${filename%.s}.o
      as "$filename" "$objname" ;; *.o ) ;;
    * )
      print "error: $filename is not a source or object file."
      exit 1 ;;
  esac
done
```

La construcción `case` en este código maneja cuatro casos. Los dos primeros son similares a los casos `if` y `elif` anteriores en este capítulo; llaman al compilador o al ensamblador si

el nombre de archivo termina en `.c` o `.s`, respectivamente.

Después, el código es un poco diferente. Recordemos que si el nombre del fichero termina en `.o` no hay que hacer nada (suponiendo que los ficheros relevantes se enlazarán más tarde). Manejamos esto con el caso `*.o )`, que no tiene declaraciones. No hay nada malo con un «caso» para el cual el script no hace nada.

Si el nombre del archivo no termina en `.o`, se produce un error. Esto se trata en el último caso, que es `*`. Esto captura cualquier cosa que no coincida con los otros casos. (De hecho, un caso `*` es análogo a un *case* por defecto en C y un caso de lo contrario en algunos lenguajes derivados de Pascal).

El bucle *for* que lo rodea procesa correctamente todos los argumentos de la línea de comandos. Esto nos lleva a otra mejora: ahora que sabemos cómo procesar todos los argumentos, deberíamos ser capaces de escribir el código que pasa todos los ficheros objeto al enlazador (el programa *ld*) al final. Podemos hacerlo construyendo una cadena de nombres de ficheros objeto, separados por espacios, y pasársela al enlazador cuando hayamos procesado todos los ficheros de entrada. Inicializamos la cadena a *null* y añadimos un nombre de fichero objeto cada vez que se crea uno, es decir, durante cada iteración del bucle *for*. El código para esto es simple, requiriendo sólo adiciones menores:

```
objfiles=""
for filename in "$@"; do
  case $filename in
    *.c )
      objname=${filename%.c}.o
      ccom "$filename" "$objname" ;;
    *.s )
      objname=${filename%.s}.o
      as "$filename" "$objname" ;;
    *.o )
      objname=$filename ;;
    * )
      print "error: $filename is not a source or object file."
      exit 1 ;;
  esac
  objfiles+=" $objname"
done
ld $objfiles
```

La primera línea en esta versión del script inicializa la variable `objfiles` a *null* [12]. Añadimos una línea de código en el caso `*.o` para establecer `objname` igual a `$filename`, porque

[12] Esto no es estrictamente necesario, porque se asume que todas las variables son nulas si no se inicializan explícitamente (a menos que la opción *nounset* esté activada). Sólo hace que el código sea más fácil de leer.

ya sabemos que es un archivo objeto. Así, el valor de `objname` se establece en todos los casos – excepto en el caso de error, en el que la rutina imprime un mensaje y se retira.

La última línea de código en el cuerpo del bucle *for* añade un espacio y el último `$objname` a `objfiles`. Llamando a este script con los mismos argumentos que en la Figura 5.1 resultaría en `$objfiles` igual a `.ª.o b.o c.o d.o"` cuando el bucle for termina (el espacio inicial no importa). Esta lista de nombres de archivos de objetos se le da a *ld* como un único argumento, pero el shell lo divide en múltiples nombres de archivo correctamente.

La Tarea 5-4 es una nueva tarea cuya solución inicial utiliza case.

Tarea 5-4

Eres un administrador de sistemas,[a] y necesitas configurar el sistema para que las variables de entorno TERM de los usuarios reflejen correctamente en qué tipo de terminal están. Escribe algún código que haga esto.

[a] Nuestras condolencias.

El código para la solución de esta tarea debe ir en el archivo */etc/profile*, que es el archivo maestro de inicio que se ejecuta para cada usuario antes de su *.profile*.

Por el momento, asumimos que tienes una configuración tradicional de tipo mainframe, en la que los terminales están cableados al ordenador. Esto significa que puedes determinar qué terminal (física) está siendo utilizada por la línea (o *tty*) en la que se encuentra. Normalmente tiene un nombre como */dev/ttyNN*, donde *NN* es el número de línea. Puedes encontrar tu tty con el comando *tty(1)*, que lo imprime en la salida estándar.

Supongamos que tu sistema tiene diez líneas más una línea de consola de sistema (*/dev/console*), con los siguientes terminales:

- Las líneas tty01, tty03 y tty04 son Givalt GL35a (nombre terminfo «gl35a»).
- La línea tty07 es una Tsoris T-2000 («t2000»).
- La línea tty08 y la consola son Shande 531s («s531»).
- El resto son Vey VT99 («vt99»).

Aquí está el código que hace el trabajo:

```
case $(tty) in
  /dev/tty0[134]            ) TERM=gl35a ;;
  /dev/tty07                ) TERM=t2000 ;;
  /dev/tty08 | /dev/console ) TERM=s531 ;;
  *                         ) TERM=vt99 ;;
```

```
esac
```

El valor que case comprueba es el resultado de la sustitución del comando. Por lo demás, la única novedad de este código es la barra OR después de /dev/tty08. Esto significa que */dev/tty08* y */dev/console* son patrones alternativos para el caso que establece TERM en «s531».

Tenga en cuenta que no es posible colocar patrones alternativos en líneas separadas a menos que utilice caracteres de continuación de barra invertida al final de todas las líneas excepto de la última. En otras palabras, la línea

```
/dev/tty08 | /dev/console ) TERM=s531 ;;
```

podría cambiarse por el ligeramente más legible:

```
/dev/tty08 | \
  /dev/console ) TERM=s531 ;;
```

La barra invertida debe estar al final de la línea. Si la omite, o si hay caracteres (incluso espacios) después de ella, el shell se queja con un mensaje de error de sintaxis.

En realidad, este problema se resuelve mejor utilizando un archivo que contenga una tabla de líneas y tipos de terminales. Veremos cómo hacerlo así en el Capítulo 7.

Cuando aparecía un caso dentro de la construcción de sustitución de comandos $(...), *ksh88* tenía un problema: la) que delimita cada patrón del código a ejecutar terminaba el $(...). Para evitarlo, era necesario poner una (delante del patrón:

```
result=$(case $input in
  ( dave ) print Dave! ;;         # Open paren required in ksh88
  ( bob ) print Bob! ;;
  esac)
```

ksh93 todavía acepta esta sintaxis, pero ya no la requiere.

5.3.1. Fusionando «cases»

A veces, al escribir una construcción de tipo case, hay instancias en las que un caso es un subconjunto de lo que debería hacerse para otro. El lenguaje C maneja esto permitiendo que un caso en un switch «continue» en el código de otro. Un hecho poco conocido es que el shell de Korn (pero no el shell de Bourne) tiene una funcionalidad similar.

Por ejemplo, supongamos que nuestro compilador de C sólo genera código ensamblador, y que depende de nuestro script front-end convertir el código ensamblador en código objeto.

En este caso, queremos pasar del caso `*.c` al caso `*.s`. Esto se hace usando `;&` para terminar el cuerpo del caso que hace la caída:

```
objfiles=""
for filename in "$@"; do
  case $filename in
    *.c )
      asmname=${filename%.c}.s
      ccom "$filename" "$asmname"
      filename=$asmname ;&      # Continuación de ejecución
    *.s )
      objname=${filename%.s}.o
      as "$filename" "$objname" ;;
    *.o )
      objname=$filename ;;
    * )
      print "error: $filename is not a source or object file."
      exit 1 ;;
  esac
  objfiles+=" $objname"
done
ld $objfiles
```

Antes de caer, el caso `*c` tiene que restablecer el valor de `filename` para que el caso `*.s` funcione correctamente. Suele ser una muy buena idea añadir un comentario indicando que la «continuación de ejecución» es intencional, aunque es más obvio en shell que en C. Volveremos a este ejemplo una vez más en el Capítulo 6 cuando discutamos cómo manejar las opciones *dash* en la línea de comandos.

5.4. select

Casi todas las construcciones de control de flujo que hemos visto hasta ahora también están disponibles en el shell de Bourne, y el shell C tiene equivalentes con diferente sintaxis. Nuestra siguiente construcción, `select`, es exclusiva del shell de Korn; además, no tiene análogos en los lenguajes de programación convencionales.

`select` le permite generar menús sencillos fácilmente. Tiene una sintaxis concisa, pero hace bastante trabajo. La sintaxis es:

```
select name [in list]
do
  declaraciones que puede usar $name ...
done
```

Es la misma sintaxis que el bucle *for* normal, excepto por la palabra clave `select`. Y al igual

que `for`, puede omitir en `list`, y por defecto será «$@», es decir, la lista de argumentos de línea de comandos entrecomillados.

Esto es lo que hace `select`:

- Genera un menú de cada elemento de la lista, formateado con números para cada opción.
- Solicita al usuario un número (con el valor de PS3)
- Almacena la opción seleccionada en el nombre de la variable y el número seleccionado en la variable incorporada `REPLY`
- Ejecuta las sentencias del cuerpo
- Repite el proceso para siempre (pero vea más abajo cómo salir)

Una vez más, un ejemplo ayudará a aclarar este proceso. Supongamos que necesita escribir el código para la Tarea 5-4, pero su vida no es tan simple. No tiene terminales conectadas a su ordenador; en su lugar, sus usuarios se comunican a través de un servidor de terminales, o se conectan remotamente, vía *telnet* o *ssh*. Esto significa, entre otras cosas, que el número de tty no determina el tipo de terminal.

Por lo tanto, no tiene otra opción que preguntar al usuario por su tipo de terminal en el momento del login. Para hacer esto, puedes poner el siguiente código en */etc/profile* (asume que tienes la misma elección de tipos de terminal):

```
PS3='terminal? '
select term in gl35a t2000 s531 vt99; do
  if [[ -n $term ]]; then
    TERM=$term
    print TERM is $TERM
    export TERM
    break
  else
    print 'invalid.'
  fi
done
```

Cuando ejecutes este código, verás este menú:

```
1) gl35a
2) t2000
3) s531
4) vt99
terminal?
```

La variable de shell incorporada `PS3` contiene la cadena de consulta que utiliza `select`; su valor por defecto es el poco útil «#? ». Así que la primera línea del código anterior la establece en un valor más relevante [13].

La sentencia `select` construye el menú a partir de la lista de opciones. Si el usuario introduce un número válido (de 1 a 4), la variable *term* se establece en el valor correspondiente; de lo contrario, es nula. (Si el usuario sólo pulsa INTRO, el shell vuelve a imprimir el menú).

El código en el cuerpo del bucle comprueba si `term` no es nulo. Si lo es, asigna `$term` a la variable de entorno `TERM`, exporta `TERM`, e imprime un mensaje de confirmación; entonces la sentencia *break* sale del bucle `select`. Si `term` es nulo, el código imprime un mensaje de error y repite el prompt (pero no el menú).

La sentencia `break` es la forma habitual de salir de un bucle `select`. En realidad (al igual que su análogo en C), se puede utilizar para salir de cualquier estructura de control circundante que hayamos visto hasta ahora (excepto `case`, donde el doble punto y coma actúa como *break*), así como de los `while` y `until` que veremos próximamente. No hemos introducido *break* hasta ahora porque algunas personas consideran que es un mal estilo de codificación utilizarlo para salir de un bucle. Sin embargo, es necesario para salir de `select` cuando el usuario hace una elección válida. [14]

Perfeccionemos nuestra solución haciendo que el menú sea más fácil de usar, para que el usuario no tenga que conocer el nombre *terminfo* de su terminal. Para ello, utilizaremos cadenas de caracteres entre comillas como elementos del menú y, a continuación, utilizaremos mayúsculas y minúsculas para determinar el nombre *terminfo*:

```
print 'Select your terminal type:'
PS3='terminal? '
select term in \
  'Givalt GL35a' \
  'Tsoris T-2000' \
  'Shande 531' \
  'Vey VT99'
do
  case $REPLY in
    1 ) TERM=gl35a ;;
    2 ) TERM=t2000 ;;
    3 ) TERM=s531 ;;
    4 ) TERM=vt99 ;;
```

[13] En cuanto a `PS1`, *ksh* realiza la sustitución de parámetros, comandos y aritmética en el valor antes de imprimirlo.

[14] Un usuario también puede teclear CTRL-D – para fin de entrada – para salir de un bucle de `selección`. Esto le da al usuario una forma uniforme de salir, pero no ayuda mucho al programador del shell.

```
    * ) print 'invalid.' ;;
  esac
  if [[ -n $term ]]; then
    print TERM is $TERM
    export TERM
    break
  fi
done
```

Este código se asemeja un poco más a una rutina de menú en un programa convencional, aunque `select` sigue proporcionando la atajo de convertir las opciones del menú en números. Enumeramos cada una de las opciones del menú en líneas separadas por razones de legibilidad, pero una vez más necesitamos caracteres de continuación para evitar que el shell se queje de la sintaxis.

Esto es lo que verá el usuario cuando se ejecute este código:

```
Select your terminal type:
1) Givalt GL35a
2) Tsoris T-2000
3) Shande 531
4) Vey VT99 terminal?
```

Este es un poco más informativo que la salida del código anterior.

Cuando se ingresa al cuerpo del bucle `select`, `$term` es igual a una de las cuatro cadenas (o es nulo si el usuario hizo una elección no válida), mientras que la variable integrada `REPLY` contiene el número que el usuario seleccionó. Necesitamos una declaración case para asignar el valor correcto a `TERM`; utilizamos el valor de `REPLY` como selector de caso.

Una vez que se completa la declaración `case`, la instrucción `if` verifica si se hizo una elección válida, como en la solución anterior. Si la elección fue válida, `TERM` ya ha sido asignado, por lo que el código simplemente imprime un mensaje de confirmación, exporta `TERM` y sale del bucle `select`. Si no fue válido, el bucle `select` repite el mensaje y vuelve a pasar por el proceso.

Dentro de un bucle `select`, si `REPLY` se establece en una cadena nula, el shell vuelve a imprimir el menú. Esto sucede, como se mencionó, cuando el usuario presiona ENTER. Pero también puedes establecer explícitamente `REPLY` en la cadena nula para forzar al shell a volver a imprimir el menú.

La variable `TMOUT` (tiempo de espera) puede afectar la instrucción `select`. Antes del bucle `select`, configúrala en algún número de segundos N y, si no se ingresa nada en ese lapso

de tiempo, el `select` se cerrará. Como se explicará más adelante, `TMOUT` también afecta al comando `read` y al mecanismo interactivo de solicitud de entrada del shell.

5.5. while y until

Las dos estructuras de control de flujo restantes que proporciona el shell de Korn son `while` y `until`. Estas son similares; ambas permiten que se ejecute repetidamente una sección de código mientras (o hasta que) se cumpla cierta condición. También se asemejan a construcciones análogas en Pascal (`while`/`do` y `repeat`/`until`) y en C (`while` y `do`/`until`).

`while` y `until` son más útiles cuando se combinan con características que veremos en el próximo capítulo, como aritmética entera, entrada/salida de variables y procesamiento de línea de comandos. Aún así, podemos mostrar un ejemplo útil incluso con las herramientas que hemos cubierto hasta ahora.

La sintaxis para `while` es:

```
while
  condición
do
  declaraciones...
done
```

Para `until`, simplemente sustituye `until` por `while` en el ejemplo anterior. Al igual que con `if`, la *condición* es realmente una lista de *declaraciones* que se ejecutan; el estado de salida de la última se utiliza como el valor de la condición. Puedes usar una condicional con `[[` y `]]` aquí, al igual que con `if`.

NOTA: La *única* diferencia entre `while` y `until` es la forma en que se maneja la condición. En `while`, el bucle se ejecuta mientras la condición sea verdadera; en `until`, se ejecuta mientras la condición sea falsa. Hasta aquí, todo es familiar. **PERO:** la condición de `until` se verifica en la parte *superior* del bucle, no en la parte *inferior* como en construcciones análogas en C y Pascal.

El resultado es que puedes convertir cualquier `until` en un `while` simplemente negando la condición. El único lugar donde `until` podría ser mejor es algo como esto:

```
until
  comando
; do
  declaraciones...
done
```

El significado de esto es esencialmente «Realizar *declaraciones* hasta que el *comando* se ejecute correctamente». En nuestra opinión, esta no es una contingencia probable. Por lo tanto, usaremos `while` en el resto de este libro.

La Tarea 5-5 es un buen candidato para `while`.

Tarea 5-5

Implementa una versión simplificada del comando `whence` incorporado en el shell.

Con «simplificada», nos referimos a que implementaremos solo la parte que verifica todos los directorios en tu **PATH** para el comando que proporcionas como argumento (no implementaremos la verificación de alias, comandos incorporados, etc.).

Podemos hacer esto seleccionando los directorios en **PATH** uno por uno, utilizando uno de los operadores de coincidencia de patrones del shell, y viendo si hay un archivo con el nombre dado en el directorio en el que tienes permisos para ejecutarlo. Aquí está el código:

```
path=$PATH
dir=${path%%:*}
while [[ -n $path ]]; do
  if [[ -x $dir/$1 &&! -d $dir/$1 ]]; then
    print "$dir/$1"
    return
  fi
  path=${path#*:}
  dir=${path%%:*}
done
return 1
```

La primera línea de este código guarda `$PATH` en **path**, nuestra propia copia temporal. Añadimos dos puntos al final para que cada directorio en **$path** termine en dos puntos (en `$PATH`, los dos puntos se usan solo *entre* directorios); el código subsiguiente depende de que esto sea así.

La siguiente línea selecciona el primer directorio de **$path** utilizando el operador que elimina la coincidencia más larga con el patrón dado. En este caso, eliminamos la coincidencia más larga con el patrón `:*`, es decir, dos puntos seguidos de cualquier cosa. Esto nos da el primer directorio en **$path**, que almacenamos en la variable `dir`.

La condición en el bucle `while` verifica si `$path` no es nulo. Si no es nulo, construye la ruta completa `$dir/$1` y verifica si hay un archivo con ese nombre para el cual tienes permisos de ejecución (y que no sea un directorio). Si es así, imprime la ruta completa y sale de la rutina con un estado de salida de 0 («OK»).

Si no se encuentra un archivo, entonces se ejecuta este código:

```
path=${path#*:}
dir=${path%%:*}
```

El primero de estos utiliza otro operador de cadena del shell: este elimina la coincidencia más corta con el patrón dado desde el principio de la cadena. En este punto, este tipo de operador debería resultarte familiar. Esta línea elimina el directorio frontal de `$path` y asigna el resultado de nuevo a **path**. La segunda línea es igual que antes del `while:` encuentra el (nuevo) directorio frontal en `$path` y lo asigna a **dir**. Esto configura el bucle para otra iteración.

Así, el código recorre todos los directorios en `PATH`. Sale cuando encuentra un archivo ejecutable coincidente o cuando ha «consumido» todo el `PATH`. Si no se encuentra ningún archivo ejecutable coincidente, no imprime nada y sale con un estado de error.

Podemos mejorar este script un poco aprovechando la utilidad UNIX *file* (1). *file* examina archivos dados como argumentos y determina qué tipo son, basándose en el *número mágico* del archivo y diversas heurísticas (conjeturas educadas). Un número mágico es un campo en el encabezado de un archivo ejecutable que el enlazador establece para identificar qué tipo de ejecutable es.

Si el *nombre de archivo* es un programa ejecutable (compilado desde C u otro lenguaje), entonces escribir `file` *`filename`* produce una salida similar a esto:

```
filename
: ELF32-bit LSB executable 80386 Version 1
```

Sin embargo, si `filename` no es un programa ejecutable, examinará las primeras líneas e intentará adivinar qué tipo de información contiene el archivo. Si el archivo contiene texto (en lugar de datos binarios), `file` buscará indicaciones de que es inglés, comandos de shell, C, Fortran, entrada *troff(1)* y varias otras cosas. A veces, `file` se equivoca, pero generalmente acierta.

Supongamos que *fred* es un archivo ejecutable en el directorio `/usr/bin`, y que *bob* es un script de shell en `/usr/local/bin`. Escribir `file /usr/bin/fred` produce esta salida:

```
/usr/bin/fred: ELF 32-bit LSB executable 80386 Version 1
```

Escribir `file /usr/local/bin/bob` tiene este resultado:

```
/usr/local/bin/bob: commands text
```

Podemos simplemente sustituir `file` por `print` para imprimir un mensaje más informativo en nuestro script:

```
path=$PATH
dir=${path%%:*}
while [[ -n $path ]]; do
  if [[ -x $dir/$1 &&! -d $dir/$1 ]]; then
    file $dir/$1
    exit
  fi
  path=${path#*:}
  dir=${path%%:*}
done
exit 1
```

Observa que al mover la declaración `dir=${path%%:*}` al principio del cuerpo del bucle, solo es necesario hacerla una vez.

Finalmente, solo para mostrar cuán pequeña es la diferencia entre `while` y `until`, señalamos que la línea:

```
until [[ ! -n $path ]]; do
```

puede usarse en lugar de:

```
while [[ -n $path ]]; do
```

con resultados idénticos.

Veremos ejemplos adicionales de *while* en el próximo capítulo.

5.5.1. break y continue

Anteriormente en este capítulo, vimos la declaración `break` utilizada con la construcción `select` para salir de un bucle. `break` se puede usar con cualquier construcción de bucle: `for, select, while` y `until`.

La declaración `continue` está relacionada; su función es omitir cualquier declaración restante en el cuerpo del bucle y comenzar la próxima iteración.

Tanto las declaraciones `break` como `continue` aceptan un argumento numérico opcional (que puede ser una expresión numérica). Esto indica cuántos bucles circundantes deben romperse o continuarse. Por ejemplo:

```
while condition1; do                # Bucle externo
  ...
  while condition2; do              # Bucle interno
    ...
```

```
    break 2                          # Sale del bucle externo
  done
done
...                                  # La ejecución continúa aquí después de break
```

Los programadores notarán que las declaraciones `break` y `continue`, especialmente con la capacidad de salir o continuar en varios niveles de bucle, compensan de manera muy limpia la falta de una palabra clave `goto` en el lenguaje de shell.

CAPÍTULO 6

OPCIONES DE LA LÍNEA DE COMANDOS Y VARIABLES TIPOGRÁFICAS

Deberías tener un sólido conocimiento de las técnicas de programación de shell ahora que has pasado por los capítulos anteriores. Lo que has aprendido hasta este punto te permite escribir muchos scripts y funciones de shell no triviales y útiles.

Sin embargo, es posible que hayas notado algunas lagunas restantes en el conocimiento que necesitas para escribir código de shell que se comporte como los comandos UNIX a los que estás acostumbrado. En particular, si eres un usuario experimentado de UNIX, puede que hayas notado que ninguno de los scripts de ejemplo mostrados hasta ahora tiene la capacidad de manejar *opciones* (precedidas por un guion (-)) en la línea de comandos. Y si programas en un lenguaje convencional como C o Pascal, habrás notado que el único tipo de datos que hemos visto en las variables de shell son cadenas de caracteres; no hemos visto cómo hacer operaciones aritméticas, por ejemplo.

Estas capacidades son ciertamente cruciales para la capacidad del shell de funcionar como un lenguaje de programación UNIX útil. En este capítulo, mostraremos cómo el shell de Korn admite estas y otras características relacionadas.

6.1. Opciones de línea de comandos

Ya hemos visto muchos ejemplos de los *parámetros posicionales* (variables llamadas **1, 2, 3,** etc.) que el shell utiliza para almacenar los argumentos de línea de comandos de un script o función de shell cuando se ejecuta. También hemos visto variables relacionadas como * (para la cadena de todos los argumentos) y # (para el número de argumentos).

De hecho, estas variables contienen toda la información de la línea de comandos del usuario. Pero considera qué sucede cuando hay opciones involucradas. Los comandos típicos de

UNIX tienen la forma `comando [- opciones] args`, lo que significa que puede haber 0 o más opciones. Si un script de shell procesa el comando **fred bob pete**, entonces `$1` es «bob» y `$2` es «pete». Pero si el comando es **fred -o bob pete**, entonces `$1` es **-o**, `$2` es «bob», y `$3` es «pete».

Podrías pensar que podrías escribir código como este para manejarlo:

```
if [[ $1 = -o ]]; then
  codigo que procesa la opcion -o
  1=$2
  2=$3
fi
procesamiento normal de $1 y $2...
```

Pero este código tiene varios problemas. En primer lugar, las asignaciones como `1=$2` son ilegales porque los parámetros posicionales son de solo lectura. Incluso si fueran legales, otro problema es que este tipo de código impone limitaciones en la cantidad de argumentos que el script puede manejar, lo cual es muy imprudente. Además, si este comando tuviera varias opciones posibles, el código para manejarlas se volvería muy desordenado muy rápidamente.

6.1.1. shift

Afortunadamente, el shell proporciona una solución a este problema. El comando `shift` realiza la función de:

```
1=$2
2=$3
...
```

para cada argumento, independientemente de cuántos haya. Si proporcionas un argumento numérico[1] a `shift`, desplazará los argumentos esa cantidad de veces; por ejemplo, `shift 3` tiene este efecto:

```
1=$4
2=$5
...
```

Esto conduce inmediatamente a un código que maneja una única opción (llamémosla `-o`) y arbitrariamente muchos argumentos:

```
if [[ $1 = -o ]]; then
    procesar la opción -o
    shift
fi
procesamiento normal de argumentos...
```

[1] En realidad, el argumento puede ser una expresión numérica; el shell la evalúa automáticamente.

Después de la construcción `if`, 1,`2,` etc., se establecen en los argumentos correctos, y # también se ajusta automáticamente.

Podemos usar `shift` junto con las características de programación que hemos visto hasta ahora para implementar esquemas de opciones simples. Sin embargo, necesitaremos ayuda adicional cuando las cosas se vuelvan más complejas. El comando integrado `getopts`, que presentaremos más adelante, proporciona esta ayuda.

`shift` por sí mismo nos da suficiente poder para implementar la opción `-N` en el script más *avanzado* que vimos en el Capítulo 4, Programación Básica de Shell (Tarea 4-1). Recuerda que este script toma un archivo de entrada que enumera artistas y la cantidad de álbumes que tienes de ellos. Ordena la lista e imprime los N números más altos, en orden descendente. El código que realiza el procesamiento real de datos es:

```
filename=$1
howmany=${2:-10}
sort -nr $filename | head -$howmany
```

Nuestra sintaxis original para llamar a este script era `highest filename [- N ]`, donde *N* se predetermina a 10 si se omite. Cambiemos esto a una sintaxis UNIX más convencional, en la que las opciones se proporcionan antes de los argumentos: `highest [- N ] filename`. Así es como escribiríamos el script con esta sintaxis:

```
if [[ $1 = -+([0-9]) ]]; then
  howmany=$1
  shift
elif [[ $1 = -* ]]; then
  print 'usage: highest [-N] filename'
  return 1
else
  howmany="-10"
  fi
  filename=$1
  sort -nr $filename | head $howmany
```

En este código, se considera que la opción se ha proporcionado si `$1` coincide con el patrón `-+([0-9])`. Esto utiliza uno de los operadores de expresiones regulares del shell de Korn, que vimos en el Capítulo 4. Observa que no rodeamos el patrón con comillas (ni siquiera comillas dobles); si lo hiciéramos, el shell lo interpretaría literalmente, no como un patrón. Este patrón significa «un guion seguido por uno o más dígitos». Si `$1` coincide, entonces lo asignamos a la variable **howmany**.

Si `$1` no coincide, comprobamos si es una opción en absoluto, es decir, si coincide con el patrón `-*`. Si es así, entonces es inválido; imprimimos un mensaje de error y salimos con

un estado de error. Si llegamos al caso final (`else`), asumimos que `$1` es un nombre de archivo y lo tratamos como tal en el código siguiente. El resto del script procesa los datos como antes.

Podemos extender lo que hemos aprendido hasta ahora a una técnica general para manejar múltiples opciones. Para concretar, supongamos que nuestro script se llama *bob* y queremos manejar las opciones `-a, -b` y `-c`:

```
while [[ $1 = -* ]]; do
  case $1 in
    -a )
      procesar opción -a ;;
    -b )
      procesar opción -b ;;
    -c )
      procesar opción -c ;;
    *)
      print 'uso: bob [-a] [-b] [-c] args...'
      return 1
  esac
  shift
done
procesamiento normal de argumentos...
```

Este código verifica repetidamente `$1` siempre que comience con un guion (-). Luego, la construcción `case` ejecuta el código apropiado según la opción `$1`. Si la opción es inválida, es decir, si comienza con un guion pero no es `-a, -b` o `-c`, el script imprime un mensaje de uso y retorna con un estado de error. Después de procesar cada opción, los argumentos se desplazan. El resultado es que los parámetros posicionales se establecen en los argumentos reales cuando el bucle `while` termina.

Observa que este código es capaz de manejar opciones de longitud arbitraria, no solo una letra (por ejemplo, `-fred` en lugar de `-a`).

6.1.2. Opciones con argumentos

Necesitamos agregar un ingrediente más para que el procesamiento de opciones sea realmente útil. Recuerda que muchos comandos tienen opciones que toman sus *propios* argumentos. Por ejemplo, el comando `cut`, en el que confiamos mucho en el Capítulo 4, acepta la opción `-d` con un argumento que determina el delimitador de campo (si no es el TAB predeterminado). Para manejar este tipo de opción, simplemente usamos otro `shift` cuando estamos procesando la opción.

Supongamos que, en nuestro script *bob*, la opción `-b` requiere su propio argumento. Aquí está el código modificado que lo procesará:

```
while [[ $1 = -* ]]; do
  case $1 in
    -a)
      procesar opción -a ;;
    -b)
      procesar opción -b $2 es el argumento de la opción
      shift ;;
    -c)
      procesar opción -c ;;
    *)
      print 'uso: bob [-a] [-b barg] [-c] args...'
      return 1
  esac
  shift
done
procesamiento normal de argumentos...
```

6.1.3. getopts

Hasta ahora, tenemos una forma completa, aunque aún limitada, de manejar las opciones de la línea de comandos. El código anterior no permite a un usuario combinar argumentos con un solo guion, por ejemplo, `-abc` en lugar de `-a -b -c`. Tampoco permite especificar argumentos para opciones sin un espacio entre ellos, por ejemplo, `-barg` además de `-b arg`[2]

El shell proporciona una forma incorporada de manejar múltiples opciones complejas sin estas restricciones. El comando incorporado `getopts`[3] se puede utilizar como condición del `while` en un bucle de procesamiento de opciones. Dada una especificación de qué opciones son válidas y cuáles requieren sus propios argumentos, configura el cuerpo del bucle para procesar cada opción por turno.

La función `getopts` toma al menos dos argumentos. El primero es una cadena que puede contener letras y dos puntos. Cada letra es una opción válida; si una letra va seguida de dos puntos, la opción requiere un argumento. Si la letra va seguida de un numeral (`#`), la opción requiere un argumento numérico. Los dos puntos (`:`) o el numeral (`#`)pueden ir seguidos de `[descripción]`, es decir, una cadena descriptiva encerrada entre corchetes que se utiliza

[2]Aunque la mayoría de los comandos de UNIX permiten esto, en realidad va en contra de las Reglas Estándar de Sintaxis de Comandos en *intro* (1) del Manual del Usuario.

[3]`getopts` reemplaza al comando externo *getopt(1)*, utilizado en la programación del shell de Bourne; `getopts` está mejor integrado en la sintaxis del shell y se ejecuta de manera más eficiente. Los programadores en C reconocerán `getopts` como muy similar a la rutina de la biblioteca estándar *getopt(3)*.

al generar mensajes de error de uso. Si añades un espacio con texto más descriptivo a la lista de caracteres de opción, ese texto también se imprime en los mensajes de error.

`getopts` selecciona opciones de la línea de comandos y asigna cada una (sin el guion inicial) a una variable cuyo nombre es el segundo argumento de `getopts`. Mientras haya opciones por procesar, `getopts` devuelve un estado de salida 0; cuando las opciones se agotan, devuelve un estado de salida 1, lo que hace que el bucle *while* salga.

De forma predeterminada, `getopts` recorre `"$@"`, la lista entre comillas de los argumentos de la línea de comandos. Sin embargo, puedes proporcionar argumentos adicionales a `getopts`, en cuyo caso los utiliza en su lugar.

`getopts` realiza algunas otras acciones que facilitan el procesamiento de opciones; las veremos a medida que examinemos cómo utilizar `getopts` en el ejemplo anterior.

```
while getopts ":ab:c" opt; do
  case $opt in
    a)
      procesar opción -a ;;
    b)
      procesar opción -b $OPTARG es el argumento de la opción ;;
    c)
      procesar opción -c ;;
    \?)
      print 'uso: bob [-a] [-b barg] [-c] args...'
      return 1
  esac
done
shift $(($OPTIND - 1))
procesamiento normal de argumentos...
```

La llamada a `getopts` en la condición del bucle `while` configura el bucle para aceptar las opciones `-a`, `-b` y `-c`, y especifica que `-b` toma un argumento. (Explicaremos el **":"** que inicia la cadena de opciones en un momento.) Cada vez que se ejecuta el cuerpo del bucle, tiene la opción más reciente disponible, sin un guion (-), en la variable `opt`.

Si el usuario escribe una opción no válida, `getopts` normalmente imprime un mensaje de error (de la forma `cmd: -o: unknown option`) y establece `opt` en `?`. `getopts` termina de procesar todas sus opciones y, si se encontró un error, el shell sale. Sin embargo, ahora aquí hay un truco oscuro: si comienzas la cadena de letras de opción con dos puntos, `getopts` no imprimirá el mensaje y el shell no saldrá. Esto te permite manejar los mensajes de error por tu cuenta.

Puedes proporcionar el colon inicial y proporcionar tu propio mensaje de error en un `case`

que maneje `?` y salga manualmente, como se indicó anteriormente, o puedes proporcionar texto descriptivo dentro de la llamada a `getopts` y dejar que el shell maneje la impresión del mensaje de error. En este último caso, el shell también saldrá automáticamente al encontrar una opción no válida.

Hemos modificado el código en la construcción `case` para reflejar lo que hace `getopts`. Pero observa que ya no hay más declaraciones `shift` dentro del bucle `while`: `getopts` no depende de `shifts` para realizar un seguimiento de dónde está. No es necesario desplazar los argumentos hasta que `getopts` haya terminado, es decir, hasta que el bucle `while` salga.

Si una opción tiene un argumento, `getopts` lo almacena en la variable `OPTARG`, que se puede usar en el código que procesa la opción.

La única declaración `shift` restante es después del bucle `while`. `getopts` almacena en la variable `OPTIND` el número del próximo argumento que se va a procesar; en este caso, ese es el número del primer argumento (no de opción) de la línea de comandos. Por ejemplo, si la línea de comandos fuera `bob -ab pete`, entonces `$OPTIND` sería «2». Si fuera `bob -a -b pete`, entonces `$OPTIND` sería «3». `OPTIND` se reinicializa a 1 cada vez que ejecutas una función, lo que te permite usar `getopts` dentro del cuerpo de una función.

La expresión `$(($OPTIND - 1))` es una expresión aritmética (como veremos más adelante en este capítulo) igual a `$OPTIND` menos 1. Este valor se utiliza como argumento para `shift`. El resultado es que la cantidad correcta de argumentos se desplaza fuera del camino, dejando los argumentos «reales» como `$1, $2,` etc.

Antes de continuar, ahora es un buen momento para resumir todo lo que hace `getopts` (incluyendo algunos puntos que aún no se han mencionado):

1. Si se le proporciona la opción `-a` y un argumento, `getopts` usa ese argumento como el nombre del programa en cualquier mensaje de error, en lugar del predeterminado, que es el nombre del script. Esto es muy útil si estás usando `getopts` dentro de una función, donde `$0` es el nombre de la función. En ese caso, es menos confuso si el mensaje de error utiliza el nombre del script en lugar del nombre de la función.

2. Su primer argumento (no de opción) es una cadena que contiene todas las letras de opción válidas. Si una opción requiere un argumento, un dos puntos sigue a su letra en la cadena. Un dos puntos inicial hace que `getopts` no imprima un mensaje de error cuando el usuario da una opción no válida. Su segundo argumento es el nombre de una variable que contiene cada letra de opción (sin ningún guion inicial) a medida

que se procesa. Al encontrar un error, esta variable contendrá un carácter ? literal.

3. Seguir una letra de opción con un # en lugar de dos puntos indica que la opción toma un argumento numérico.

4. Cuando una opción toma un argumento (la letra de opción va seguida de dos puntos o de un símbolo #), agregar un signo de interrogación indica que el argumento de la opción es opcional (es decir, no es obligatorio).

5. Si se proporcionan argumentos adicionales en la línea de comandos de `getopts` después de la cadena de opciones y el nombre de la variable, se utilizan en lugar de `$@`.

6. Si una opción toma un argumento, el argumento se almacena en la variable `OPTARG`.

7. La variable `OPTIND` contiene un número igual al próximo argumento de la línea de comandos que se va a procesar. Después de que `getopts` haya terminado, es igual al número del primer argumento «real».

8. Si el primer carácter en la cadena de opciones es + (o el segundo carácter después de dos puntos iniciales), entonces las opciones también pueden comenzar con +. En este caso, la variable de opción tendrá un valor que comienza con +.

`getopts` puede hacer mucho, mucho más de lo que se ha descrito aquí. Consulta el Apéndice B, que proporciona la historia completa.

Las ventajas de `getopts` son que minimiza el código adicional necesario para procesar opciones y admite completamente la sintaxis estándar de opciones UNIX (según se especifica en *intro(1)* del Manual del Usuario).

Como ejemplo más concreto, volvamos a nuestro front-end del compilador de C (Tarea 4-2). Hasta ahora, le hemos dado a nuestro script la capacidad de procesar archivos fuente de C (que terminan en .c), archivos de código ensamblador (.s) y archivos de código objeto (.o). Aquí está la última versión del script:

```
objfiles=""
for filename in "$@"; do
  case $filename in
    *.c)
      objname=${filename%.c}.o
      compile $filename $objname ;;
    *.s)
      objname=${filename%.s}.o
      assemble $filename $objname ;;
```

```
    *.o)
      objname=$filename ;;
    *)
      print "error: $filename is not a source or object file."
      return 1 ;;
  esac
  objfiles="$objfiles $objname"
done
ld $objfiles
```

Ahora podemos darle al script la capacidad de manejar opciones. Para saber qué opciones necesitaremos, tendremos que discutir más a fondo lo que hacen los compiladores.

Más sobre compiladores C

El compilador de C en un sistema UNIX moderno típico (ANSI C en System V Release 4) tiene aproximadamente 30 opciones diferentes de línea de comandos, pero nos limitaremos a las más ampliamente utilizadas.

Esto es lo que implementaremos. Todos los compiladores proporcionan la capacidad de eliminar el paso final de enlace, es decir, la llamada al enlazador *ld*. Esto es útil para compilar código C en archivos de código objeto que se vincularán más tarde y para aprovechar la verificación de errores del compilador por separado antes de intentar vincular. La opción `-c` suprime el paso de enlace, produciendo solo los archivos de código objeto compilados.

Los compiladores de C también son capaces de incluir mucha información adicional en un archivo de código objeto que puede ser utilizada por un depurador (aunque es ignorada por el enlazador y el programa en ejecución). Si no sabes qué es un depurador, consulta el Capítulo 9, Depuración de Programas de Shell. El depurador necesita mucha información sobre el código C original para poder hacer su trabajo; la opción `-g` indica al compilador que incluya esta información en su salida de código objeto.

Si aún no estás familiarizado con los compiladores de C de UNIX, es posible que te haya parecido extraño ver en el último capítulo que el enlazador coloca su salida (el programa ejecutable) en un archivo llamado `a.out`. Esta convención es un vestigio histórico que a nadie le ha importado cambiar. Aunque ciertamente es posible cambiar el nombre del ejecutable con el comando *mv*, el compilador de C proporciona la opción `-o` *filename*, que utiliza `filename` en lugar de `a.out`.

Otra opción que admitiremos aquí tiene que ver con las *bibliotecas*. Una biblioteca es una colección de código objeto, parte del cual se incluirá en el ejecutable en tiempo de

enlace. (Esto contrasta con un archivo de código objeto precompilado, que se vincula por completo). Cada biblioteca incluye una gran cantidad de código objeto que respalda un cierto tipo de interfaz o actividad; los sistemas UNIX típicos tienen bibliotecas para cosas como redes, funciones matemáticas y gráficos.

Las bibliotecas son extremadamente útiles como bloques de construcción que ayudan a los programadores a escribir programas complejos sin tener que «reinventar la rueda» cada vez. La opción `-l` *name* del compilador de C indica al enlazador que incluya el código necesario de la biblioteca *name*.[4] en el ejecutable que construye. Una biblioteca particular llamada c (el archivo `libc.a`) siempre está incluida. Esto se conoce como la biblioteca de tiempo de ejecución de C; contiene código para la capacidad de entrada y salida estándar de C, entre otras cosas.

Finalmente, es posible que un buen compilador de C realice ciertas cosas que hagan que su código objeto de salida sea más pequeño y eficiente. Colectivamente, estas cosas se llaman *optimización*. Puedes pensar en un *optimizador* como un paso adicional en el proceso de compilación que analiza el código objeto de salida y lo cambia para mejor. La opción `-O` invoca al optimizador.

La Tabla 6.1 resume las opciones que incorporaremos en nuestro front-end del compilador de C.

Tabla 6.1: Opciones Populares del Compilador de C

Opción	Significado
-c	Producir solo código objeto; no invocar al enlazador
-g	Incluir información de depuración en archivos de código objeto
-l *lib*	Incluir la biblioteca *lib* al enlazar
-o *exefile*	Producir el archivo ejecutable *exefile* en lugar del predeterminado `a.out`
-O	Invocar al optimizador

También debes tener en cuenta esta información sobre las opciones:

- Las opciones **-o** y **-l lib** se pasan simplemente al enlazador *(ld)*, que las procesa por sí mismo.
- En la mayoría de los sistemas, `ld` requiere que las opciones de biblioteca vayan después de los ficheros objeto en la línea de órdenes. (Además, el orden de las bibliotecas en la línea de órdenes es importante. Si una rutina en `libA.a` hace referencia a otra rutina de `libB.a`, entonces `libA.a` debe aparecer primero en la línea de comandos

[4]En realidad, esto es un archivo llamado *lib* `name .a` en un directorio de bibliotecas estándar como `/lib`.

(`-lA -lB`). Esto implica que la biblioteca C (`libc.a`) debe cargarse en último lugar, ya que las rutinas de otras bibliotecas casi siempre dependen de las rutinas estándar de la biblioteca C.

- La opción **-l** *lib* se puede usar varias veces para vincular varias bibliotecas.
- La opción **-g** se pasa al comando `ccom` (el programa que realiza la compilación real de C).
- Supondremos que el optimizador es un programa independiente llamado *optimize* que acepta un archivo de código objeto como argumento y lo optimiza «en su lugar», es decir, sin producir un archivo de salida separado.

Para nuestro front-end, hemos elegido dejar que el shell se encargue de imprimir el mensaje de uso. Aquí está el código para el script *occ* que incluye el procesamiento de opciones:

```
# inicializar variables relacionadas con las opciones
do_link=true
debug=""
link_libs="-l c"
exefile=""
opt=false

# procesar opciones de línea de comandos
while getopts ":cgl:o:O" opt; do
  case $opt in
    c )
      do_link=false ;;
    g )
      debug="-g" ;;
    l )
      link_libs="$link_libs -l $OPTARG" ;;
    o )
      exefile="-o $OPTARG" ;;
    O )
      opt=true ;;
    \? )
      print 'uso: occ [-cgO] [-l lib] [-o file] files...'
      return 1 ;;
  esac
done
shift $(($OPTIND - 1))

# procesar los archivos de entrada
objfiles=""
for filename in "$@"; do
  case $filename in
    *.c )
      objname=${filename%.c}.o
```

```
      ccom $debug $filename $objname
      if [[ $opt = true ]]; then
        optimize $objname
      fi ;;
    *.s )
      objname=${filename%.s}.o
      as $filename $objname ;;
    *.o )
      objname=$filename ;;
    * )
      print "error: $filename no es un archivo fuente u objeto."
      return 1 ;;
  esac
  objfiles="$objfiles $objname"
done
if [[ $do_link = true ]]; then
  ld $exefile $link_libs $objfiles
fi
```

Examinemos la parte del código que procesa las opciones. Las primeras líneas inicializan variables que utilizaremos más adelante para almacenar el estado de cada una de las opciones. Utilizamos «true» y «false» como valores de verdad para mayor legibilidad; son simplemente cadenas y, de lo contrario, no tienen un significado especial. Las inicializaciones reflejan estas suposiciones:

1. Querremos realizar el enlace.
2. No querremos que el compilador genere información de depuración que consuma espacio.
3. La única biblioteca de código objeto que necesitaremos es *c*, la biblioteca estándar de tiempo de ejecución de C que se enlaza automáticamente. *E*l archivo ejecutable que crea el enlazador será el archivo predeterminado del enlazador, *a.out*. *N*o querremos invocar al optimizador.

Las construcciones `while, getopts` y `case` procesan las opciones de la misma manera que en el ejemplo anterior. Esto es lo que hace el código que maneja cada opción:

- Si se proporciona la opción `-c`, se establece la bandera `do_link` en «false», lo que hará que la condición `if` al final del script sea falsa, lo que significa que el enlazador no se ejecutará.
- Si se proporciona `-g`, la variable `debug` se establece en «*-g*». Esto se pasa en la línea de comandos al compilador.

- Cada `-l` *lib* que se proporciona se agrega a la variable `link_libs`, de modo que cuando el bucle `while` sale, `$link_libs` es toda la cadena de opciones `-l`. Esta cadena se pasa al enlazador.
- Si se proporciona `-o` *file*, la variable `exefile` se establece en «*-o file*». Esta cadena se pasa al enlazador.
- Si se especifica `-O`, se establecerá la bandera `opt`. Esta especificación hace que la condición condicional `if [[ $opt = true ]]` sea verdadera, lo que significa que se ejecutará el optimizador.

El resto del código es una modificación del bucle `for` que ya hemos visto; las modificaciones son resultados directos del procesamiento de opciones anterior y deberían ser autoexplicativas.

6.2. Variables Enteras y Aritmética

La expresión `$(($OPTIND - 1))` en el último ejemplo da una pista de cómo el shell puede realizar aritmética entera. Como podrías imaginar, el shell interpreta las palabras rodeadas por `$(( y ))` como expresiones aritméticas. Las variables en expresiones aritméticas no necesitan ir precedidas por signos de dólar, aunque no está mal hacerlo.

Las expresiones aritméticas se evalúan dentro de comillas dobles, al igual que las tildes, variables y sustituciones de comandos. *Finalmente*, estamos en posición de establecer la regla definitiva sobre citar cadenas: cuando haya dudas, encierra una cadena entre comillas simples, a menos que contenga tildes o cualquier expresión que involucre un signo de dólar, en cuyo caso deberías usar comillas dobles.

Por ejemplo, el comando *date (1)* en versiones derivadas de System V de UNIX acepta argumentos que le indican cómo formatear su salida. El argumento `+%j` le indica que imprima el día del año, es decir, el número de días desde el 31 de diciembre del año anterior.

Podemos usar `+%j` para imprimir un pequeño mensaje de anticipación a las vacaciones:

```
print "Solo quedan $(( (365-$(date +%j)) / 7 )) semanas hasta el Año Nuevo!"
```

Mostraremos dónde encaja esto en el esquema general del procesamiento de la línea de comandos en el Capítulo 7.

La función de expresiones aritméticas está integrada en la sintaxis del shell de Korn y estaba

disponible en el shell de Bourne (en la mayoría de las versiones) solo a través del comando externo *expr(1)*. Así que es otro ejemplo de una característica deseable proporcionada por un comando externo (es decir, un truco sintáctico) que se integra mejor en el shell. `[[...]]` y *getopts* son también ejemplos de esta tendencia de diseño.

Mientras que `expr` y *ksh88* estaban limitados a la aritmética entera, *ksh93* admite la aritmética de punto flotante. Como veremos en breve, puedes realizar prácticamente cualquier cálculo en el shell de Korn que podrías hacer en C o en la mayoría de los otros lenguajes de programación.

Los operadores aritméticos del shell de Korn son equivalentes a sus contrapartes en el lenguaje C. La precedencia y la asociatividad son las mismas que en C. (Más detalles sobre la compatibilidad del shell de Korn con el lenguaje C se pueden encontrar en el Apéndice B; dichos detalles son de interés principalmente para personas que ya están familiarizadas con C.) La Tabla 6.2 muestra los operadores aritméticos que se admiten, en orden de mayor a menor precedencia. Aunque algunos de ellos son (o contienen) caracteres especiales, no es necesario escaparlos con barra invertida, porque están dentro de la sintaxis `$((...))`.

Tabla 6.2: Operadores Aritméticos

Operador	**Significado**	**Asociatividad**
`++ --`	Incremento y decremento, prefijo y postfijo	De izquierda a derecha
`+ - ! ~`	Más y menos unario; negación lógica y bit a bit	De derecha a izquierda
`**`	Exponenciación [5]	De derecha a izquierda
`* / %`	Multiplicación, división y resta	De izquierda a derecha
`+ -`	Adición y sustracción	De izquierda a derecha
`<< >>`	Desplazamiento de bits a izquierda y derecha	De izquierda a derecha
`< <= > >=`	Comparaciones	De izquierda a derecha
`== =!`	Iguales y no iguales	De izquierda a derecha
`&`	AND a nivel de bits	De izquierda a derecha
`^`	OR exclusivo a nivel de bits	De izquierda a derecha
`|`	OR a nivel de bits	De izquierda a derecha
`&&`	AND lógico (cortocircuito)	De izquierda a derecha
`||`	OR lógico (cortocircuito)	De izquierda a derecha
`?:`	Expresión condicional	De derecha a izquierda
`= =+ -= *= /=%= &= ^= <<= >>=`	Operadores de asignación	De derecha a izquierda
`,`	Evaluación secuencial	De izquierda a derecha

Los paréntesis se pueden utilizar para agrupar subexpresiones. La sintaxis de la expresión aritmética (como en C) admite operadores relacionales como «valores de verdad» de 1 para

[5] *ksh93m* y más reciente. El operador ** no está en lenguaje C.

verdadero y 0 para falso.

Por ejemplo, `$((3 > 2))` tiene el valor 1; `$(( (3 > 2) || (4 <= 1) ))` también tiene el valor 1, ya que al menos una de las dos subexpresiones es verdadera.

Si estás familiarizado con C, C++ o Java, los operadores enumerados en la Tabla 6.2 te resultarán familiares. Pero si no lo estás, algunos de ellos merecen una pequeña explicación.

Las formas de asignación de los operadores regulares son una forma conveniente de actualizar una variable de manera más compacta. Por ejemplo, en Pascal o Fortran podrías escribir `x = x + 2` para sumar 2 a x. El `+=` te permite hacerlo de manera más concisa: `$((x += 2))` suma 2 a x y almacena el resultado nuevamente en x. (Compara esto con la reciente adición del operador `+=` a *ksh93* para la concatenación de cadenas).

Dado que sumar y restar 1 son operaciones tan frecuentes, los operadores `++` y `--` proporcionan una forma aún más abreviada de realizarlas. Como podrías imaginar, `++` suma 1, mientras que `--` resta 1. Estos son operadores unarios. Echemos un vistazo rápido a cómo funcionan.

```
$ i=5
$ print $((i++)) $i
5 6
$ print $((++i)) $i
7 7
```

¿Qué está sucediendo aquí? En ambos casos, el valor de i se incrementa en uno. Pero el valor devuelto por el operador depende de su ubicación con respecto a la variable en la que se está operando. Un operador de sufijo (que ocurre después de la variable) devuelve el valor *antiguo* de la variable como resultado de la expresión y luego incrementa la variable. Por el contrario, un operador de `prefijo`, que viene antes de la variable, incrementa la variable primero y luego devuelve el nuevo valor. El operador `--` funciona de la misma manera que `++`, pero disminuye la variable en uno en lugar de incrementarla.

El shell también admite números en base `N`, donde `N` puede ser hasta 64. La notación `B#N` significa «N base B». Por supuesto, si omites el `B#`, la base predetermina a 10. Los dígitos son `0-9, a-z (10-35), A-Z (36-61), @ (62)` y `_(63)`. (Cuando la base es menor o igual a 36, puedes usar letras en mayúsculas y minúsculas). Por ejemplo:

```
$ print el número ksh 43#G es $((43#G))
el número ksh 43#G es 42
```

Curiosamente, puedes usar variables de shell para contener subexpresiones, y el shell sustituye el valor de la variable al realizar la aritmética. Por ejemplo:

```
$ almost_the_answer=10+20
$ print $almost_the_answer
10+20
$ print $(( almost_the_answer + 12 ))
42
```

6.2.1. Funciones Aritméticas Incorporadas

El shell proporciona varias funciones aritméticas y trigonométricas incorporadas para su uso con `$((...))`. Se llaman utilizando la sintaxis de llamada a función en C. Las funciones trigonométricas esperan que los argumentos estén en radianes, no en grados. (Hay 2 πradianes en un círculo). Por ejemplo, recordando los días de la escuela secundaria, recuerda que 45 grados es πdivido por 4. Digamos que necesitamos el coseno de 45 grados:

```
$ pi=3.1415927 # Valor aproximado para pi
$ print el coseno de pi/4 es $(( cos(pi / 4) ))
el coseno de pi/4 es 0.707106772982
```

Una mejor aproximación de πse puede obtener utilizando la función atan:

```
pi=$(( 4. * atan(1.) )) # Un valor mejor para pi
```

La Tabla 6.3 enumera las funciones aritméticas incorporadas.

Tabla 6.3: Funciones aritméticas incorporadas

Función	**Retorna**	**Función**	**Retorna**
abs	Valor absoluto	hypot	Distancia euclidiana
acos	Arco coseno	int	Parte entera
asin	Arco seno	log	Logaritmo natural
atan	Arco tangente	pow	Exponeciación (x^y)
atan2	Arco tangente de dos variables	sin	Seno
cos	Coseno	sinh	Seno hiperbólico
cosh	Coseno hiperbólico	sqrt	Raíz cuadrada
exp	Exponencial (e^x)	tan	Tangente
fmod	Resto en coma flotante	tanh	Tangente hiperbólica

NOTA: `hypot pow atan2` y `fmod` fueron agregados a partir de *ksh93e*.

6.2.2. Condiciones Aritméticas

Otro constructo, estrechamente relacionado con `$((...))`, es `((...))` (sin el signo de dólar inicial). Usamos esto para evaluar pruebas de condiciones aritméticas, al igual que `[[...]]` se utiliza para pruebas de cadenas, atributos de archivos y otros tipos de pruebas.

`((...))` evalúa operadores relacionales de manera diferente a `$((...))` para que puedas usarlo en las construcciones `if` y `while`. En lugar de producir un resultado textual, simple-

mente establece su estado de salida según la verdad de la expresión: 0 si es verdadera, 1 en caso contrario. Entonces, por ejemplo, `((3 > 2))` produce un estado de salida 0, al igual que `(( (3 > 2) || (4 <= 1) ))`, pero `(( (3 > 2) && (4 <= 1) ))` tiene un estado de salida 1 ya que la segunda subexpresión no es verdadera.

También puedes usar valores numéricos como valores de verdad dentro de este constructo. Es similar al concepto análogo en C, lo que significa que es un tanto contraintuitivo para los programadores que no son de C: un valor de 0 significa *falso* (es decir, devuelve un estado de salida 1), y un valor no igual a 0 significa *verdadero* (devuelve un estado de salida 0), por ejemplo, `(( 14 ))` es verdadero. Consulta el código del depurador *kshdb* en el Capítulo 9 para ver dos ejemplos más de esto.

6.2.3. Variables y Asignación Aritmética

El constructo `(( ... ))` también se puede utilizar para definir variables enteras y asignarles valores. La declaración:

```
(( var=expression ))
```

crea la variable entera intvar (si aún no existe) y le asigna el resultado de la *expresión*.

La sintaxis de doble paréntesis es la recomendada. Sin embargo, si prefiere utilizar un comando para realizar operaciones aritméticas, el shell le proporciona uno: el comando incorporado `let`. La sintaxis es:

```
let var=expression
```

No es necesario (porque es redundante) rodear la expresión con `$(( y ))` en una declaración `let`. Como con cualquier asignación de variables, no debe haber espacio a ambos lados del signo igual (=). Es una buena práctica rodear las expresiones con comillas, ya que muchos caracteres son tratados como especiales por el shell (por ejemplo, * , # , y paréntesis); además, debes poner entre comillas expresiones que incluyan espacios en blanco (espacios o TAB). Consulta la Tabla 6.4 para ejemplos.

```
let "x = 3.1415927" "y = 1.41421"
```

Mientras que *ksh88* solo te permitía usar variables enteras, *ksh93* ya no tiene esta restricción y las variables también pueden ser de punto flotante. (Un *número entero* es lo que se llamaba un «número entero» en la escuela, un número que no tiene una parte fraccionaria, como 17 o 42. Los números de punto flotante, en cambio, pueden tener partes fraccionarias, como 3.1415927). El shell busca un punto decimal para determinar que un valor es de punto

flotante. Sin él, los valores se tratan como enteros. Esto es principalmente un problema para la división: la división entera truncará cualquier parte fraccionaria. El operador % requiere un divisor entero.

El shell proporciona dos alias integrados para declarar variables numéricas: `integer` para variables enteras y `float` para variables de punto flotante. (Ambos son alias para el comando `typeset` con diferentes opciones. Se proporcionan más detalles en la Sección 6.5.3, más adelante en este capítulo).

Finalmente, todas las asignaciones tanto a variables enteras como de punto flotante se evalúan automáticamente como expresiones aritméticas. Esto significa que no es necesario utilizar el comando `let`:

```
$ integer the_answer
$ the_answer=12+30
$ print $the_answer
42
```

Tabla 6.4: Asignaciones de Expresiones Enteras de Muestra

Asignación let x=	**Valor $x**
x=1+4	5
'x = 1 + 4'	5
'x = 1.234 + 3'	4.234
'x = (2+3) * 5'	25
'x = 2 + 3 * 5'	17
'x = 17 / 3'	5
'x = 17 / 3.0'	5.66666666667
'17% 3'	2
'1<<4'	16
'48>>3'	6
'17 & 3'	1
'17 & 3'	19
'17 ^ 3'	18

La Tarea 6-1 es una pequeña tarea que hace uso de la aritmética.

Tarea 6-1

Escribe un script llamado *pages* que, dado el nombre de un archivo de texto, indique cuántas páginas de salida contiene. Supón que hay 66 líneas por página, pero proporciona una opción que permita al usuario anular eso.

Haremos que nuestra opción sea *-N*, al estilo de *head*. La sintaxis para esta única opción es tan simple que no necesitamos molestarnos con `getopts`. Aquí está el código:

```
if [[ $1 = -+([0-9]) ]]; then
  let page_lines=${1#-}
  shift
else
  let page_lines=66
fi

let file_lines="$(wc -l < $1)"

let pages=file_lines/page_lines
if (( file_lines % page_lines > 0 )); then
  let pages=pages+1
fi

print "$1 tiene $pages páginas de texto."
```

Observa que utilizamos la condición entera `(( file_lines % page_lines > 0 ))` en lugar de la forma `[[ ... ]]`.

En el corazón de este código se encuentra la utilidad de UNIX *wc(1)*, que cuenta el número de líneas, palabras y caracteres (bytes) en su entrada. De forma predeterminada, su salida se ve algo así:

```
8 34 161 bob
```

La salida de *wc* significa que el archivo bob tiene 8 líneas, 34 palabras y 161 caracteres. *wc* reconoce las opciones `-l, -w` y `-c`, que le indican que imprima solo el número de líneas, palabras o caracteres, respectivamente.

Normalmente, *wc* imprime el nombre de su archivo de entrada (dado como argumento). Como solo queremos el número de líneas, tenemos que hacer dos cosas. Primero, le proporcionamos la entrada mediante la redirección de archivos, como en `wc -l < bob` en lugar de `wc -l bob`. Esto produce el número de líneas precedido por uno o más espacios.

Desafortunadamente, ese espacio complica las cosas: la declaración `let file_lines=$(wc -l < $1)` se convierte en `let file_lines= N` después de la sustitución de comandos; el espacio después del signo igual es un error. Eso nos lleva a la segunda modificación, las comillas alrededor de la expresión de sustitución de comandos. La declaración `let file_lines="N"` es perfectamente legal, y `let` sabe cómo eliminar el espacio inicial.

La primera cláusula `if` en el script *pages* verifica si se proporcionó una opción y, si es así, elimina el guion (-) y lo asigna a la variable `page_lines`. *wc* en la expresión de sustitución

de comandos devuelve el número de líneas en el archivo cuyo nombre se proporciona como argumento.

El siguiente grupo de líneas calcula el número de páginas y, si hay un resto después de la división, suma 1. Finalmente, se imprime el mensaje correspondiente.

Como un ejemplo más grande de aritmética entera, completaremos nuestra emulación de las funciones `pushd` y `popd` del shell C (Tarea 4-7). Recuerda que estas funciones operan en `DIRSTACK`, una pila de directorios representada como una cadena con los nombres de directorios separados por espacios. `pushd` y `popd` del shell C toman tipos adicionales de argumentos, que son:

- `pushd +n` toma el *n-ésimo* directorio en la pila (comenzando con 0), lo rota hacia arriba y hace `cd` a él.
- `pushd` sin argumentos, en lugar de quejarse, intercambia los dos directorios principales en la pila y hace `cd` al nuevo directorio superior.
- `popd +n` toma el *n-ésimo* directorio en la pila y simplemente lo elimina.

La característica más útil de estas funciones es la capacidad de acceder al n-ésimo directorio en la pila. Aquí tienes las versiones más recientes de ambas funciones:

```
function pushd {                          # apila el directorio actual en la pila
  dirname=$1
  if [[ -d $dirname && -x $dirname ]]; then
    cd $dirname
    DIRSTACK="$dirname ${DIRSTACK:-$PWD}"
    print "$DIRSTACK"
  else
    print "aún en $PWD."
  fi
}

function popd {                           # desapila el directorio de la pila, realiza cd al
    nuevo tope
  if [[ -n $DIRSTACK ]]; then
    DIRSTACK=${DIRSTACK#* }
    cd ${DIRSTACK%% *}
    print "$PWD"
  else
    print "pila vacía, aún en $PWD."
  fi
}
```

Para acceder al *n-ésimo* directorio, utilizamos un bucle `while` que transfiere el directorio superior a una copia temporal de la pila n veces. Pondremos el bucle en una función llamada

`getNdirs` que se ve así:

```
function getNdirs {
  stackfront=''
  let count=0
  while (( count < $1 )); do
    stackfront="$stackfront ${DIRSTACK%% *}"
    DIRSTACK=${DIRSTACK#* }
    let count=count+1
  done
}
```

El argumento pasado a `getNdirs` es el *n* en cuestión. La variable `stackfront` es la copia temporal que contendrá los primeros n directorios cuando el bucle esté completo. `stackfront` comienza como nulo; `count`, que cuenta el número de iteraciones del bucle, comienza como 0.

La primera línea del cuerpo del bucle agrega la parte superior de la pila (`${DIRSTACK%% *}`) a `stackfront`; la segunda línea elimina la parte superior de la pila. La última línea incrementa el contador para la próxima iteración. Todo el bucle se ejecuta *N* veces, para valores de `count` de 0 a *N*-1.

Cuando el bucle termina, el último directorio en `$stackfront` es el *n-ésimo* directorio. La expresión `${stackfront##*}` extrae este directorio. Además, `DIRSTACK` ahora contiene la «parte posterior» de la pila, es decir, la pila sin los primeros *n* directorios. Con esto en mente, ahora podemos escribir el código para las versiones mejoradas de `pushd` y `popd`:

```
function pushd {
  if [[ $1 = ++([0-9]) ]]; then
    # caso de pushd +n: rotar el n-ésimo directorio hacia arriba
    let num=${1#+}
    getNdirs

    $num newtop=${stackfront##* }
    stackfront=${stackfront%$newtop}

    DIRSTACK="$newtop $stackfront $DIRSTACK"
    cd $newtop

  elif [[ -z $1 ]]; then
    # caso de pushd sin argumentos; intercambiar los dos directorios superiores
    firstdir=${DIRSTACK%% *}
    DIRSTACK=${DIRSTACK#* }
    seconddir=${DIRSTACK%% *}
    DIRSTACK=${DIRSTACK#* }
    DIRSTACK="$seconddir $firstdir $DIRSTACK"
    cd $seconddir
```

```
  else
    cd $dirname
    # caso normal de pushd con nombre de directorio
    dirname=$1
    if [[ -d $dirname && -x $dirname ]]; then
      DIRSTACK="$dirname ${DIRSTACK:-$PWD}"
      print "$DIRSTACK"
    else
      print "aún en "$PWD."
    fi
  fi
}

function popd {
  # sacar el directorio de la pila, cd al nuevo superior
  if [[ $1 = ++([0-9]) ]]; then
    # caso de popd +n: eliminar el n-ésimo directorio de la pila
    let num={$1#+}
    getNdirs $num
    stackfront=${stackfront% *}
    DIRSTACK="$stackfront $DIRSTACK"
  else
    # caso normal de popd sin argumento
    if [[ -n $DIRSTACK ]]; then
      DIRSTACK=${DIRSTACK#* }
      cd ${DIRSTACK%% *}
      print "$PWD"
    else
      print "pila vacía, aún en $PWD."
    fi
  fi
}
```

Las funciones han crecido bastante; veámoslas por separado. La instrucción `if` al principio de `pushd` verifica si el primer argumento es una opción en la forma de $+N$. Si es así, se ejecuta el primer bloque de código. El primer `let` simplemente elimina el signo más (+) del argumento y asigna el resultado, como un número entero, a la variable `num`. Esto, a su vez, se pasa a la función `getNdirs`.

Las siguientes dos instrucciones de asignación establecen `newtop` como el directorio N -*ésimo*, es decir, el último directorio en `$stackfront`, y eliminan ese directorio de `stackfront`. Las dos últimas líneas en esta parte de `pushd` vuelven a armar la pila en el orden apropiado y realizan `cd` al nuevo directorio superior.

La cláusula `elif` verifica si no hay argumento, en cuyo caso `pushd` debería intercambiar los dos directorios superiores en la pila. Las primeras cuatro líneas de esta cláusula asignan los dos directorios superiores a `firstdir` y `seconddir`, y los eliminan de la pila. Luego, como

se mencionó anteriormente, el código vuelve a armar la pila en el nuevo orden y realiza `cd` al nuevo directorio superior.

La cláusula `else` corresponde al caso habitual, donde el usuario proporciona un nombre de directorio como argumento.

`popd` funciona de manera similar. La cláusula `if` verifica la opción + *N*, que en este caso significa eliminar el *N -ésimo* directorio. Un `let` extrae el *N* como un número entero; la función `getNdirs` coloca los primeros *N* directorios en `stackfront`.
Luego, la línea `stackfront=${stackfront%*}` elimina el último directorio (el *N -ésimo* directorio) de `stackfront`. Finalmente, se vuelve a armar la pila con el *N -ésimo* directorio faltante.

La cláusula `else` cubre el caso habitual, donde el usuario no proporciona un argumento.

Antes de dejar este tema, aquí tienes algunos ejercicios que deberían poner a prueba tu comprensión de este código:

1. Agrega código a `pushd` para que salga con un mensaje de error si el usuario no proporciona un argumento y la pila contiene menos de dos directorios.
2. Verifica que cuando el usuario especifica + *N* y *N* supera el número de directorios en la pila, tanto `pushd` como `popd` utilicen el último directorio como el *N -ésimo* directorio.
3. Modifica la función `getNdirs` para que verifique la condición anterior y salga con un mensaje de error apropiado si es verdadero.
4. Cambia `getNdirs` para que use `cut` (con sustitución de comandos), en lugar del bucle `while`, para extraer los primeros *N* directorios. Esto utiliza menos código pero se ejecuta más lentamente debido a los procesos adicionales generados.

6.3. for aritmétrico

El bucle `for` descrito en el Capítulo 5 ha estado en los shells Unix desde la versión 7 del Bourne Shell. Como se ha mencionado, no puede hacer bucles al estilo Pascal o C para un número fijo de iteraciones con ese bucle `for`. *ksh93* introdujo el bucle aritmético *for* para remediar esta situación y para acercar el shell a un lenguaje de programación tradicional (algunos dirían «real»).

La sintaxis se parece a las facilidades aritméticas del shell que acabamos de ver. Es casi idéntica a la sintaxis del bucle `for` de C, excepto por el conjunto extra de paréntesis:

```
for ((init; condition; increment))
do
  sentencias ...
done
```

Aquí, *init* representa algo que debe hacerse una vez, al comienzo del bucle. La *condición* se comprueba, y mientras sea verdadera, el shell ejecuta las *sentencias*. Antes de volver al principio del bucle para comprobar la condición de nuevo, el shell ejecuta *increment*.

Cualquier *init, condition* e *increment* puede ser omitida. Una condición *omitida* se trata como *verdadera*; es decir, el cuerpo del bucle siempre se ejecuta. (El llamado «bucle infinito» requiere que utilice algún otro método para salir del bucle). Utilizaremos el bucle aritmético *for* para la Tarea 6-2, que es nuestra siguiente tarea, bastante simple.

Tarea 6-2

Escribe un script simple que tome una lista de números en la línea de comandos y los sume, imprimiendo el resultado.

Aquí está el código; la explicación viene a continuación:

```
sum=0
count=$#
for ((i = 1;i <= count; i++))
do
  let "sum += $1"
  shift
done
print $sum
```

La primera línea inicializa la variable `sum` a 0. `sum` acumula la suma de los argumentos. La segunda línea es más que nada para facilitar la lectura; `count` indica cuántos argumentos hay. La tercera línea inicia el bucle propiamente dicho. La variable `i` es la variable de control del bucle. La cláusula *init* la pone a 1, la cláusula *condition* la comprueba con el límite de `count`, y la cláusula `increment` le añade 1 cada vez que se pasa por el bucle. Una cosa que notará de inmediato es que dentro del encabezado del bucle `for`, no hay necesidad del $ delante del nombre de una variable para obtener el valor de esa variable. Esto es cierto para cualquier expresión aritmética en el shell de Korn.

El cuerpo del bucle hace la suma. Simplemente deja que haga las cuentas: la suma se consigue añadiendo `$1` al valor de `sum`. El comando *shift* mueve el siguiente argumento a

`$1` para usarlo en la siguiente vuelta del bucle. Finalmente, cuando el bucle termina, el script imprime el resultado.

El bucle *for* aritmético es particularmente útil para trabajar con todos los elementos de un array indexado, que veremos en la siguiente sección.

6.4. Arrays (matrices)

Hasta ahora, hemos visto tres tipos de variables: cadenas de caracteres, enteros y números de punto flotante. El cuarto tipo de variable que admite el shell de Korn es un `array`. Como probablemente sepas, un array es como una lista de cosas; puedes referirte a elementos específicos en un array con *índices*, de modo que `a[i]` se refiere al *i-ésimo* elemento del array a. El shell de Korn proporciona dos tipos de *arrays*: *arrays* indexados y *arrays* asociativos.

6.4.1. Matrices indexadas

El shell de Korn proporciona una funcionalidad de arrays indexados que, aunque útil, es mucho más limitada que las características análogas en lenguajes de programación convencionales. En particular, los arrays indexados solo pueden ser unidimensionales (es decir, no hay arrays de arrays) y están limitados a 4096 elementos.[6] Los índices comienzan en 0. Esto implica que el valor máximo del índice es 4095. Además, los índices pueden ser cualquier expresión aritmética: *ksh* evalúa automáticamente la expresión para obtener el índice real.

Hay tres formas de asignar valores a elementos de un array. La primera es la más intuitiva: puedes usar la sintaxis estándar de asignación de variables del shell con el índice del array entre corchetes ([]). Por ejemplo:

```
nicknames[2]=bob
nicknames[3]=ed
```

Estas asignaciones colocan los valores `bob` y `ed` en los elementos del array `nicknames` con índices 2 y 3, respectivamente. Al igual que con las variables regulares del shell, los valores asignados a elementos del array se tratan como cadenas de caracteres a menos que la asignación esté precedida por `let`, o el array se haya declarado numérico con una de las opciones `typeset -i, -ui, -E` o `-F`. (Estrictamente hablando, el valor asignado con *let*

[6] 4096 es un valor mínimo en *ksh93*. Las versiones recientes permiten hasta 64K elementos.

sigue siendo una cadena; es solo que con *let*, el shell evalúa la expresión aritmética que se asigna para producir esa cadena).

La segunda forma de asignar valores a un array es con una variante de la declaración set, que vimos en el Capítulo 3. La declaración:

```
set -A aname val1 val2 val3 ...
```

crea el array `aname` (si aún no existe) y asigna *val1* a `aname[0]`, *val2* a `aname[1]`, etc. Como podrías imaginar, esto es más conveniente para cargar un array con un conjunto inicial de valores.

La tercera forma (recomendada) es usar la forma de asignación compuesta:

```
aname=(val1 val2 val3)
```

A partir de *ksh93j*, puedes usar el operador `+=` para agregar valores a un array:

```
aname+=(val4 val5 val6)
```

Para extraer un valor de un array, utiliza la sintaxis `${aname[i]}`. Por ejemplo, `${nicknames[2]}` tiene el valor «bob». El índice *i* puede ser una expresión aritmética, como se explicó anteriormente. Si usas * o @ en lugar del índice, el valor será todos los elementos, separados por espacios. Omitir el índice ([$nicknames}) es lo mismo que especificar el índice 0 (`${nicknames[0]}`).

Ahora llegamos al aspecto algo inusual de los arrays en el shell de Korn. Supongamos que los únicos valores asignados a *nicknames* son los dos que vimos anteriormente. Si escribes `print "${nicknames[*]}"`, verás la salida:

```
bob ed
```

En otras palabras, `nicknames[0]` y `nicknames[1]` no existen. Además, si escribieras:

```
nicknames[9]=pete
nicknames[31]=ralph
```

y luego escribieras *print "${nicknames[*]}"*, la salida se vería así:

```
bob ed pete ralph
```

Esto explica por qué dijimos «los elementos de *nicknames* con índices 2 y 3» anteriormente, en lugar de «el segundo y tercer elemento de *nicknames*». Cualquier elemento de array con valores no asignados simplemente no existe; si intentas acceder a sus valores, obtienes cadenas nulas.

Puedes preservar cualquier espacio en blanco que coloques en los elementos de tu array usando `"${aname[@]}"` (con las comillas dobles) en lugar de `${aname[*]}`, al igual que puedes hacerlo con `"$@"` en lugar de `$*` o `"$*"`.

El shell proporciona un operador que te dice cuántos elementos tiene definidos un array: `${#aname[*]}`. Así que `${#nicknames[*]}` tiene el valor 4. Ten en cuenta que necesitas el `[*]` porque el nombre del array solo se interpreta como el elemento 0. Esto significa, por ejemplo, que `${#nicknames}` es igual a la longitud de `nicknames[0]` (ver Capítulo 4). Dado que `nicknames[0]` no existe, el valor de `${#nicknames}` es 0, la longitud de la cadena nula.

Si piensas en un array como un mapeo de enteros a valores (es decir, ingresas un número y obtienes un valor), puedes ver por qué los arrays son estructuras de datos «dominadas por números». Debido a que las tareas de programación del shell están mucho más orientadas hacia cadenas de caracteres y texto que hacia números, la facilidad de array indexado del shell no es tan ampliamente útil como podría parecer al principio.

Sin embargo, podemos encontrar cosas útiles para hacer con arrays indexados. Aquí tienes una solución más limpia para la Tarea 5-4, en la que un usuario puede seleccionar su tipo de terminal (variable de entorno TERM) al iniciar sesión. Recuerda que la versión «amigable para el usuario» de este código usaba `select` y un enunciado `case`:

```
print 'Selecciona tu tipo de terminal:'
PS3='¿Terminal? '
select term in \
  'Givalt GL35a' \
  'Tsoris T-2000' \
  'Shande 531' \
  'Vey VT99'
do
  case $REPLY in
    1 ) TERM=gl35a ;;
    2 ) TERM=t2000 ;;
    3 ) TERM=s531 ;;
    4 ) TERM=vt99 ;;
    * ) print "inválido." ;;
  esac
  if [[ -n $term ]]; then
    print "TERM es $TERM"
    export TERM
    break
  fi
done
```

Podemos eliminar todo el enunciado `case` aprovechando el hecho de que el enunciado

`select` almacena la elección numérica del usuario en la variable `REPLY`. Solo necesitamos una línea de código que almacene todas las posibilidades para `TERM` en un array, en un orden que corresponda a los elementos en el menú `select`. Luego podemos usar `$REPLY` para indexar el array. El código resultante es:

```
set -A termnames gl35a t2000 s531 vt99
print 'Selecciona tu tipo de terminal:'
PS3='¿Terminal? '
select term in \
  'Givalt GL35a' \
  'Tsoris T-2000' \
  'Shande 531' \
  'Vey VT99'
do
  if [[ -n $term ]]; then
    TERM=${termnames[REPLY-1]}
    print "TERM es $TERM"
    export TERM
    break
  fi
done
```

Este código configura el array `termnames` para que `${termnames[0]}` sea *gl35a*, `${termnames[1]}` sea *t2000*, etc. La línea `TERM=${termnames[REPLY-1]}` reemplaza esencialmente todo el enunciado `case` utilizando `REPLY` para indexar el array.

Observa que el shell sabe interpretar el texto en un índice de array como una expresión aritmética, como si estuviera encerrado en `((` y `))`, lo que a su vez significa que la variable no necesita ir precedida por un signo de dólar (`$`). Debemos restar 1 al valor de `REPLY` porque los índices de array comienzan en 0, mientras que los números de elementos del menú `select` comienzan en 1.

Piensa en cómo podrías usar arrays para mantener la pila de directorios para `pushd` y `popd`. El bucle `for` aritmético también podría resultar útil.

6.4.2. Matrices asociativas

Como se mencionó en la sección anterior, las tareas de programación en shell suelen ser orientadas a cadenas de texto en lugar de a números. *ksh93* introdujo *arrays asociativos* en el shell para mejorar la programabilidad de la misma. Los arrays asociativos son fundamentales en la programación en lenguajes como *awk, perl* y *python.*

Un array asociativo es un array indexado por valores de cadena. Proporciona una asociación entre el índice de cadena y el valor del array en ese índice, lo que hace que ciertos tipos

de tareas funcionen de manera mucho más natural. Le indicas al shell que un array es asociativo usando `typeset -A`:

```
typeset -A person
person[firstname]="frank"
person[lastname]="jones"
```

Podemos reescribir nuestro ejemplo de terminal de la sección anterior utilizando arrays asociativos:

```
typeset -A termnames                    # termnames es un array asociativo
termnames=([Givalt GL35a]=gl35a         # Rellenar los valores
           [Tsoris T-2000]=t2000
           [Shande 531]=s531
           [Vey VT99]=vt99)
print 'Selecciona tu tipo de terminal:'
PS3='¿Terminal? '
select term in "${!termnames[@]}"       # Presentar menú de índices de array
do
  if [[ -n $term ]]; then
    TERM=${termnames[$term]}           # Usar cadena para indexar el array
    print "TERM es $TERM"
    break
  fi
done
```

Observa que los corchetes cuadrados en la asignación compuesta actúan como comillas dobles; aunque está bien poner entre comillas los índices de cadena, no es necesario. También observa la construcción `${!termnames[@]}`. Es un poco complicado, pero nos da todos los índices del array como cadenas separadas entre comillas que conservan cualquier espacio en blanco incrustado, al igual que `$@` (ver la siguiente sección).

A partir de *ksh93j*, al igual que para los arrays regulares, puedes usar el operador `+=` para agregar valores a un array asociativo:

```
termnames+=([Boosha B-27]=boo27 [Cherpah C-42]=chc42)
```

Como nota adicional, si aplicas `typeset -A` a una variable previamente existente que no era un array, el valor actual de esa variable se colocará en el índice 0. La razón es que el shell trata `$x` como equivalente a `${x[0]}`, de modo que si haces:

```
x=fred
typeset -A x
print $x
```

todavía obtendrás `fred`.

6.4.3. Operadores de nombre de matriz

En el Capítulo 4 vimos que el shell proporciona numerosas formas de acceder y manipular los valores de las variables del shell. En particular, vimos operadores que funcionan con *nombres* de variables del shell. Varios operadores adicionales se aplican a los arrays. Se describen en la Tabla 6.5.

Tabla 6.5: Operadores relacionados con nombres de arrays

Operador	Significado
`${!array[subíndice]}`	Devuelve el subíndice real utilizado para indexar el array. Los subíndices pueden provenir de expresiones aritméticas o de los valores de variables del shell.
`${!array[*]}`	Lista de todos los subíndices del array asociativo.
`${!array[@]}`	Lista de todos los subíndices de la matriz asociativa, pero se expande a palabras separadas cuando se utiliza dentro de comillas dobles.

Puedes pensar en la construcción `${!...}` para producir el array real como conceptualmente similar a su uso con variables `nameref`. Allí, indica la variable real a la que se refiere una `nameref`. Con arrays, produce el subíndice real utilizado para acceder a un elemento en particular. Esto es valioso porque los subíndices se pueden generar dinámicamente, por ejemplo, como expresiones aritméticas o mediante las diversas operaciones de cadena disponibles en el shell. Aquí tienes un ejemplo sencillo:

```
$ set -A letters a b c d e f g h i j k l m n o p q r s t u v w x y z
$ print ${letters[20+2+1]}
x
$ print ${!letters[20+2+1]}
23
```

Para recorrer todos los elementos de un array indexado, podrías usar fácilmente un bucle `for` aritmético que fuera de 0 a, por ejemplo, `${#letters[*]}` (el número de elementos en `letters`). Los arrays asociativos son diferentes: no hay límites inferiores o superiores en los índices del array, ya que todos son cadenas. Los dos últimos operadores en la Tabla 6.5 facilitan recorrer un array asociativo:

```
typeset -A bob                          # Crear un array asociativo
...                                     # Rellénalo
for index in "${!bob[@]}"; do           # Para todos los subíndices de bob
  print "bob[$index] es ${bob[$index]}" # Imprimir cada elemento
  ...
done
```

Analógamente a la diferencia entre `$*` y `"$@"`, es mejor usar la versión con `@` del operador, dentro de comillas dobles, para preservar los valores originales de cadena. (Usamos

`${!var[@]}` con `select` en el último ejemplo en la sección anterior sobre arrays asociativos).

6.5. typeset

La última característica del shell de Korn que se relaciona con los tipos de valores que pueden contener las variables es el comando `typeset`. Si eres programador, podrías suponer que `typeset` se utiliza para especificar el *tipo* de una variable (entero, cadena, etc.); estarías parcialmente en lo correcto. `typeset` es una colección bastante *ad hoc* de cosas que puedes hacer a las variables para restringir los tipos de valores que pueden tomar. Las operaciones se especifican mediante opciones para `typeset`; la sintaxis básica es:

```
typeset opción nombre_variable[=valor]
```

Aquí, *opción* es una letra de opción precedida por un guion o signo más. Las opciones se pueden combinar y se pueden usar varios *nombre_variable*. Si omites *nombre_variable*, el shell imprime una lista de variables para las cuales se activa la opción dada.

Las opciones disponibles se dividen en dos categorías básicas:

- Operaciones de formato de cadena, como justificación a la derecha e izquierda, truncamiento y control de mayúsculas y minúsculas.
- Funciones de tipo y atributo que son de interés principal para programadores avanzados.

6.5.1. Variables locales en funciones

`typeset` sin opciones tiene un significado importante: si se utiliza una declaración `typeset` dentro de la definición de una función, las variables involucradas se vuelven *locales* a esa función (además de cualquier propiedad que puedan adquirir como resultado de las opciones de `typeset`). La capacidad de definir variables que son locales a unidades de «subprogramas» (procedimientos, funciones, subrutinas, etc.) es necesaria para escribir programas grandes, porque ayuda a mantener los subprogramas independientes del programa principal y entre sí.

NOTA: Los nombres de variables locales están restringidos a identificadores simples. Cuando se usa `typeset` con un nombre de variable compuesto (es decir, uno que contiene puntos), esa variable se vuelve automáticamente global, incluso si la declaración `typeset` ocurre

dentro del cuerpo de una función.

Si solo deseas declarar una variable local en una función, utiliza `typeset` sin opciones. Por ejemplo:

```
function afunc {
  typeset diffvar
  samevar=funcvalue
  diffvar=funcvalue
  print "samevar es $samevar"
  print "diffvar es $diffvar"
}

samevar=globvalue
diffvar=globvalue
print "samevar es $samevar"
print "diffvar es $diffvar"
afunc
print "samevar es $samevar"
print "diffvar es $diffvar"
```

Este código imprimirá lo siguiente:

```
samevar es globvalue
diffvar es globvalue
samevar es funcvalue
diffvar es funcvalue
samevar es funcvalue
diffvar es globvalue
```

La Figura 6.1 muestra esto gráficamente.

Verás varios ejemplos adicionales de variables locales dentro de funciones en el Capítulo 9.

6.5.2. Opciones de formato de cadena

Ahora veamos las diversas opciones de `typeset`. La Tabla 6.6 enumera las opciones de formato de cadena; las tres primeras toman un argumento numérico opcional.

Aquí tienes algunos ejemplos simples. Supongamos que la variable `alpha` se asigna con las letras del alfabeto, en mayúsculas y minúsculas alternadas, rodeadas por tres espacios a cada lado:

```
alpha="  aBcDeFgHiJkLmNoPqRsTuVwXyZ  "
```

La Tabla 6.7 muestra algunas declaraciones `typeset` y sus valores resultantes (suponiendo que cada una de las declaraciones se ejecute «independientemente»).

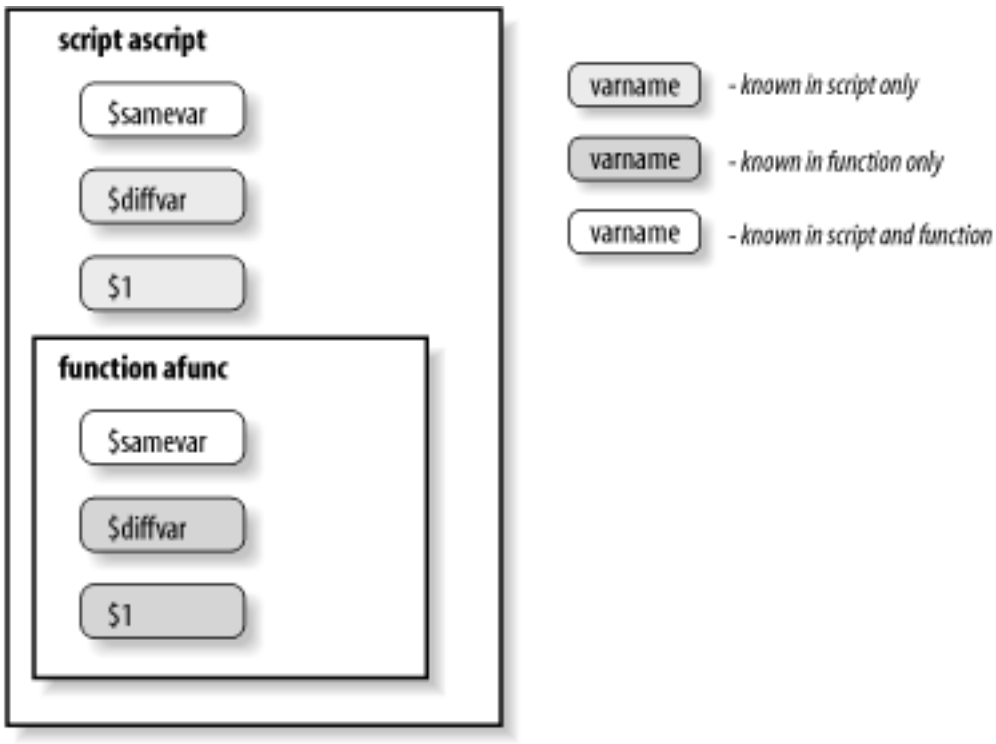

Figura 6.1: Variables locales en funciones

Tabla 6.6: Opciones de formato de cadena de `typeset`

Opción	**Operación**
`-Ln`	Justificar a la izquierda. Elimina los espacios iniciales; si se da n, rellena con espacios o trunca a la derecha hasta la longitud n.
`-Rn`	Justificar a la derecha. Elimina los espacios finales; si se da n, rellena con espacios o trunca a la izquierda hasta la longitud n.
`-Zn`	Si se utiliza con -R, añade 0 a la izquierda en lugar de espacios si es necesario. Si se utiliza con -L, elimina los 0 iniciales. Por sí mismo, actúa igual que -RZ.
`-l`	Convierte las letras a minúsculas.
`-u`	Convierte las letras a mayúsculas.

Cuando ejecutas `typeset` en una variable existente, su efecto es *acumulativo* con cualquier `typeset` que se haya utilizado anteriormente. Esto tiene excepciones obvias:

- Un `typeset -u` deshace un `typeset -l`, y viceversa.
- Un `typeset -R` deshace un `typeset -L`, y viceversa.
- No puedes combinar `typeset -l` y `typeset -u` con algunos de los atributos numéricos, como `typeset -E`. Sin embargo, ten en cuenta que `typeset -ui` crea enteros sin signo.
- `typeset -A` y `typeset -n` (array asociativo y referencias de nombre, respectivamente) no son acumulativos.

Tabla 6.7: Ejemplos de opciones de formato de cadena de `typeset`

Declaración	Valor de *v*
`typeset -L v=$alpha`	`''aBcDeFgHiJkLmNoPqRsTuVwXyZ  ''`
`typeset -L10 v=$alpha`	`''aBcDeFgHiJ''`
`typeset -R v=$alpha`	`''       aBcDeFgHiJkLmNoPqRsTuVwXyZ''`
`typeset -R16 v=$alpha`	`''kLmNoPqRsTuVwXyZ''`
`typeset -l v=$alpha`	`''   abcdefghijklmnopqrstuvwxyz   ''`
`typeset -uR5 v=$alpha`	`''VWXYZ''`
`typeset -Z8 v="123.50"`	`''00123.50''`

Puedes desactivar explícitamente las opciones de `typeset` escribiendo `typeset +o`, donde `o` es la opción que activaste anteriormente. Por supuesto, es difícil imaginar escenarios en los que desees activar y desactivar múltiples opciones de formato de `typeset` una y otra vez; generalmente, configuras una opción de `typeset` en una variable dada solo una vez.

Una aplicación evidente para las opciones `-L` y `-R` es aquella en la que necesitas una salida de ancho fijo. La fuente más ubicua de salida de ancho fijo en el sistema Unix se refleja en la Tarea 6-3.

Tarea 6-3

Fingir que `ls` no hace una salida de varias columnas; escribe un script de shell que lo haga.

Por simplicidad, supongamos que nuestra versión de Unix es antigua y que los nombres de archivo están limitados a 14 caracteres.[7]

Nuestra solución para esta tarea se basa en muchos de los conceptos que hemos visto anteriormente en este capítulo. También se basa en el hecho de que `set -A` (para construir arrays) se puede combinar con la sustitución de comandos de una manera interesante: cada palabra (separada por espacios, tabulaciones o saltos de línea) se convierte en un elemento del array. Por ejemplo, si el archivo *bob* contiene 50 palabras, el array *fred* tiene 50 elementos después de la declaración:

```
set -A fred $(< bob)
```

Nuestra estrategia es obtener los nombres de todos los archivos en el directorio dado en una variable de array. Utilizamos un bucle `for` aritmético, como vimos anteriormente en este capítulo, para obtener cada nombre de archivo en una variable cuya longitud se ha establecido en 14. Imprimimos esa variable en formato de cinco columnas, con dos espacios

[7] No conocemos sistemas Unix modernos que aún tengan esta restricción. Pero aplicarla aquí simplifica considerablemente el problema de programación.

entre cada columna (para un total de 80 columnas), usando un contador para llevar un registro de las columnas. Aquí está el código:

```
set -A filenames $(ls $1)
typeset -L14 fname
let numcols=5

for ((count = 0; count < ${#filenames[*]} ; count++)); do
  fname=${filenames[count]}
  print -rn "$fname "
  if (( (count+1) % numcols == 0 )); then
    print     # nueva línea
  fi
done

if (( count % numcols != 0 )); then
  print
fi
```

La primera línea configura el array `filenames` para contener todos los archivos en el directorio dado por el primer argumento (el directorio actual por defecto). La declaración `typeset` configura la variable `fname` para tener un ancho fijo de 14 caracteres. La siguiente línea inicializa `numcols` con el número de columnas por línea.

El bucle *for* aritmético itera una vez por cada elemento en `filenames`. En el cuerpo del bucle, la primera línea asigna el próximo elemento del array a la variable de ancho fijo. La declaración `print` imprime este último seguido de dos espacios; la opción `-n` suprime la última nueva línea de `print`.

Luego está la declaración `if`, que determina cuándo comenzar la próxima línea. Verifica el resto de `(count + 1) % numcols` - recuerda que los signos de dólar no son necesarios dentro de la construcción `$((...))` - y si el resultado es 0, es hora de emitir una nueva línea a través de una declaración `print` sin argumentos. Observa que aunque `$count` aumenta en 1 con cada iteración del bucle, el resto pasa por un ciclo de 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, ...

Después del bucle, una construcción `if` emite una nueva línea final si es necesario, es decir, si el `if` dentro del bucle no lo acaba de hacer.

También podemos usar opciones de `typeset` para mejorar el código de nuestro script *dosmv* (Tarea 5-3), que traduce nombres de archivo en un directorio dado del formato MS-DOS al formato Unix. El código del script es:

```
for filename in ${1:+$1/}* ; do
  newfilename=$(print $filename | tr '[A-Z]' '[a-z]')
  newfilename=${newfilename%.}
```

```
  # sutileza: entrecomillar el valor de $newfilename para hacer una comparación de cadenas,
  # no una coincidencia de expresiones regulares
  if [[ $filename != "$newfilename" ]]; then
    print "$filename -> $newfilename"
    mv $filename $newfilename
  fi
done
```

Podemos reemplazar la llamada a `tr` en el bucle `for` con una a `typeset -l` antes del bucle:

```
typeset -l newfilename
for filename in ${1:+$1/}* ; do
  newfilename=${filename%.}
  # sutileza: entrecomillar el valor de $newfilename para hacer una comparación de cadenas,
  # no una coincidencia de expresiones regulares
  if [[ $filename != "$newfilename" ]]; then
    print "$filename -> $newfilename"
    mv $filename $newfilename
  fi
done
```

De esta manera, la traducción a minúsculas se realiza automáticamente cada vez que se asigna un valor a *newfilename*. No solo este código es más limpio, sino que también es más eficiente, ya que se eliminan los procesos adicionales creados por ‘tr‘ y la sustitución de comandos.

6.5.3. Opciones de tipo y atributo

Las otras opciones de `typeset` son más útiles para programadores avanzados de shell que están «ajustando» scripts extensos. Estas opciones se enumeran en la Tabla 6.8.

Tabla 6.8: Opciones de tipo y atributo de `typeset`

Opción	**Operación**
-A	Crear un array asociativo
-En	Representa la variable internamente como un número de punto flotante de doble precisión; mejora la eficiencia de la aritmética de punto flotante. Si se indica *n*, es el número de cifras significativas que se utilizarán en la salida. Los números grandes se imprimen en notación científica: `[-]d.ddde±dd`. Los números pequeños se imprimen en notación normal: `[-]ddd.ddd`.

Opción	Operación
-Fn	Representa la variable internamente como un número de punto flotante de doble precisión; mejora la eficiencia de la aritmética de punto flotante. Si se da *n*, es el número de decimales a utilizar en la salida. Todos los valores se imprimen en notación regular: `[-]ddd.ddd`.
-H	En sistemas que no son Unix, los nombres de archivo de estilo Unix se convierten al formato apropiado para el sistema local.
-in	Representa la variable internamente como un entero; mejora la eficiencia de la aritmética de enteros. Si se da *n*, es la base utilizada para la salida. La base por defecto es 10.
-n	Crea una variable *nameref* (ver Capítulo 4).
-p	Cuando se utiliza por sí mismo, imprime declaraciones compuestas (`typeset`) que describen los atributos de cada una de las variables del shell que tienen atributos establecidos. Con opciones adicionales, solo imprime aquellas variables que tienen el atributo correspondiente establecido. Destinado para volcar el estado del shell en un archivo que luego puede ser utilizado por un shell diferente para recrear el estado original del shell.
-r	Hace que la variable sea de solo lectura: prohíbe la asignación a la misma y no permite que sea *desestablecida*. El comando integrado `readonly` hace lo mismo, pero `readonly` no puede ser usado para variables locales.
-t	«Etiqueta» la variable. La lista de variables etiquetadas está disponible mediante `typeset +t`. Esta opción está obsoleta.
-uin	Representa la variable internamente como un entero *sin signo*. Esto se discute más adelante en el Apéndice B. Si se proporciona *n*, es la base utilizada para la salida. La base predeterminada es 10.[8]
-x	Esto hace lo mismo que el comando `export`, pero `export` no puede ser utilizado para variables locales.
-f	Hace referencia solo a nombres de funciones; consulte la Sección 6.5.4, más adelante en este capítulo.

La opción `-i` es la más útil. Puedes agregarla a un script cuando hayas terminado de escribir y depurar para hacer que la aritmética se ejecute un poco más rápido, aunque la mejora de velocidad solo será evidente si tu script realiza *mucha* aritmética. La palabra

más legible `integer` es un alias integrado para `typeset -i`, de modo que `integer x=5` es lo mismo que `typeset -i x=5`. De manera similar, la palabra `float` es un alias predefinido para `typeset -E`.[9].

La opción `-r` es útil para configurar «constantes» en scripts de shell; las constantes son como variables, excepto que no puedes cambiar sus valores una vez que se han inicializado. Las constantes te permiten dar nombres a los valores incluso si no deseas que se cambien; se considera una buena práctica de programación utilizar constantes con nombres en programas grandes.

La solución para la Tarea 6-3 contiene un buen candidato para `typeset -r`: la variable `numcols`, que especifica el número de columnas en la salida. Dado que `numcols` es un entero, también podríamos usar la opción `-i`, es decir, reemplazar `let numcols=5` con `typeset -ri numcols=5`. Si intentáramos asignar otro valor a `numcols`, el shell respondería con el mensaje de error `ksh: numcols: is read only`.

Estas opciones también son útiles sin argumentos, es decir, para ver qué variables existen que tienen esas opciones activadas.

6.5.4. Opciones de función

La opción `-f` tiene varias subopciones, todas relacionadas con funciones. Estas se enumeran en la Tabla 6.9.

Tabla 6.9: Opciones de función de `typeset`

Opción	Operación
-f	Sin argumentos, imprime todas las definiciones de funciones.
-f *fname*	Imprime la definición de la función *fname*.
+f	Imprime todos los nombres de funciones.
-ft	Activa el modo de rastreo para la(s) función(es) nombrada(s). (Capítulo 9)
+ft	Desactiva el modo de rastreo para la(s) función(es) nombrada(s). (Capítulo 9)
-fu	Define los nombres dados como funciones de carga automática. (Capítulo 4)

Dos de estas tienen alias integrados que son más mnemotécnicos: `functions` (nota la s) es un alias para `typeset -f` y `autoload` es un alias para `typeset -fu`.

[9]Los programadores de C, C++ y Java pueden encontrar sorprendente la elección de la palabra *float*, ya que internamente el shell usa números de punto flotante de doble precisión. Teorizamos que se eligió el nombre *float* porque su significado es más obvio para el no programador que la palabra *double*

[9]Esta función sólo está disponible en *ksh93m* y posteriores.

Finalmente, si escribes `typeset` sin argumentos, verás una lista de todas las variables que tienen atributos establecidos (en un orden no discernible), precedidas por las palabras clave apropiadas para cualquier opción de `typeset` que esté activada. Por ejemplo, escribir `typeset` en un shell sin personalizar te da una lista de la mayoría de las variables integradas del shell y sus atributos que se ve así:

```
export HISTFILE
integer TMOUT
export FCEDIT
export _AST_FEATURES
export TERM
HISTEDIT
PS2
PS3
integer PPID
export MAIL
export LOGNAME
export EXINIT
integer LINENO
export PATH
integer HISTCMD
export _
export OLDPWD
export PWD
float precision 3 SECONDS
export SHELL
integer RANDOM
export HOME
export VISUAL
export MANPATH
export EDITOR
export ENV
export HISTSIZE
export USER
export LANG
export MORE
integer OPTIND
integer MAILCHECK
export CDPATH
readonly namespace .sh
```

Aquí está la salida de `typeset -p`:

```
typeset -x HISTFILE
typeset -i TMOUT
typeset -x FCEDIT
typeset -x _AST_FEATURES
typeset -x TERM
typeset -x ASIS_DIR
typeset HISTEDIT
typeset PS2
```

```
typeset PS3
typeset -i PPID
typeset -x MAIL
typeset -x LOGNAME
typeset -x EXINIT
typeset -i LINENO
typeset -x PATH
typeset -i HISTCMD
typeset -x _
typeset -x OLDPWD
typeset -x PWD
typeset -F 3 SECONDS
typeset -x SHELL
typeset -i RANDOM
typeset -x HOME
typeset -x VISUAL
typeset -x MANPATH
typeset -x EDITOR
typeset -x ENV
typeset -x HISTSIZE
typeset -x USER
typeset -x LANG
typeset -x MORE
typeset -i OPTIND
typeset -i MAILCHECK
typeset -x CDPATH
typeset -r .sh
```

El siguiente comando guarda los valores y atributos de todas las variables del shell para un uso posterior:

```
{ set ; typeset -p ;} > varlist
```

CAPÍTULO 7

ENTRADA/SALIDA Y PROCESAMIENTO DESDE LA LÍNEA DE COMANDOS

Las últimas series de capítulos se han sumergido en detalles sobre diversas técnicas de programación en el shell, enfocándose principalmente en el flujo de datos y control a través de los programas en el shell. En este capítulo, cambiaremos el enfoque hacia dos temas relacionados. El primero es el mecanismo del shell para realizar entrada y salida orientada a archivos. Presentaremos información que amplía lo que ya sabes sobre los redireccionadores básicos de E/S del shell.

En segundo lugar, nos centraremos en la E/S a nivel de línea y palabra. Este es un tema fundamentalmente diferente, ya que implica mover información entre los dominios de archivos/terminales y variables del shell. `print` y la sustitución de comandos son dos formas de hacer esto que ya hemos visto.

Nuestra discusión sobre la E/S de línea y palabra nos llevará a una explicación más detallada de cómo el shell procesa las líneas de comandos. Esta información es necesaria para que puedas entender exactamente cómo el shell maneja las *citas* y para que puedas apreciar el poder de un comando avanzado llamado `eval`, que cubriremos al final del capítulo.

7.1. Redireccionadores de E/S

En el Capítulo 1 aprendiste sobre los redireccionadores básicos de E/S del shell, `<`, `>`, y `|`. Aunque estos son suficientes para cubrir el 95 % de tu vida en Unix, debes saber que el shell de Korn admite un total de 20 redireccionadores de E/S. La Tabla 7.1 los enumera, incluyendo los tres que ya hemos visto. Aunque algunos de los demás son útiles, otros son principalmente para programadores de sistemas. Esperaremos hasta el próximo capítulo para discutir los últimos tres, que, junto con `>|` y `<<<`, no están presentes en la mayoría de

las versiones del shell de Bourne.

Tabla 7.1: Redireccionadores de E/S

Redireccionador	**Función**
> file	Dirigir la salida estándar al archivo
< file	Tomar entrada estándar del archivo
cmd1 | cmd2	Tubería; tomar salida estándar de cmd1 como entrada estándar a cmd2
>> file	Dirigir la salida estándar al archivo; añadir al archivo si ya existe
>| file	Fuerza la salida estándar a fichero incluso si *noclobber* está activado
<> file	Abrir archivo tanto para lectura como para escritura en la entrada estándar[1]
<< label	*here-document*; ver texto
<< − label	*here-document* variante; ver texto
<<< label	*here-document* cadena (string); ver texto
n> file	Duplicar la entrada estándar del descriptor de archivo n
n< file	Establecer archivo como descriptor de archivo de entrada n
<&n	Duplicar la entrada estándar del descriptor de archivo n
>&n	Duplicar salida estándar a descriptor de fichero n
<&n-	Mover el descriptor de fichero n a la entrada estándar
>&n-	Mover el descriptor de fichero n a la salida estándar
<&-	Cerrar la entrada estándar
>&-	Cerrar la salida estándar
|&	Proceso en segundo plano con E/S desde el shell padre
n<&p	Mover la entrada del coproceso al descriptor de fichero n
n>&p	Mover la salida del coproceso al descriptor de fichero n

Observa que algunos de los redireccionadores en la Tabla 7.1 contienen un dígito *n* y que sus descripciones contienen el término «descriptor de archivo»; cubriremos eso en un momento. (De hecho, cualquier redireccionador que comience con < o > puede usarse con un descriptor de archivo; esto se omite en la tabla por simplicidad).

Los dos primeros redireccionadores nuevos, `>>` y `>|`, son variaciones simples del redireccionador de salida estándar `>`. `>>` agrega al archivo de salida (en lugar de sobrescribirlo) si ya existe; de lo contrario, actúa exactamente como `>`. Un uso común de `>>` es agregar una línea a un archivo de inicialización (como `.profile` o `.mailrc`) cuando no quieres molestias con un editor de texto. Por ejemplo:

```
$ cat >> .mailrc
> alias fred frederick@longmachinename.longcompanyname.com
> ^D
$
```

[1] Normalmente, los archivos abiertos con < se abren en modo de solo lectura.

Como vimos en el Capítulo 1, `cat` sin argumentos utiliza la entrada estándar como su entrada. Esto te permite escribir la entrada y finalizarla con CTRL-D en su propia línea. La línea de `alias` se agregará al archivo `.mailrc` si ya existe; si no existe, se crea el archivo con esa única línea.

Recuerda del Capítulo 3 que puedes evitar que el shell sobrescriba un archivo con `> file` escribiendo `set -o noclobber`. El operador `>|` anula `noclobber`, es el redireccionador de «¡Hazlo de todos modos, demonios!».

Los sistemas Unix te permiten abrir archivos en modo solo lectura, solo escritura y lectura/escritura. El redireccionador `<` abre el archivo de entrada en modo solo lectura; si un programa intenta escribir en la entrada estándar, recibirá un error. De manera similar, el redireccionador `>` abre el archivo de salida en modo solo escritura; intentar leer desde la salida estándar genera un error. El redireccionador `<>` abre un archivo para lectura y escritura, por defecto en la entrada estándar. Depende del programa invocado darse cuenta de esto y aprovecharlo, pero es útil en el caso en que un programa pueda querer actualizar datos en un archivo «en su lugar». Este operador se utiliza principalmente para escribir clientes de red; consulta la Sección 7.1.4, más adelante en este capítulo, para ver un ejemplo.

7.1.1. Here-Documents

El redireccionador `<< label` básicamente fuerza que la entrada a un comando sea el texto del programa del shell, que se lee hasta que hay una línea que contiene solo la *etiqueta*. La entrada en el medio se llama un *here-document* (documento aquí). Los *here-documents* no son muy interesantes cuando se usan desde el indicador de comandos. De hecho, es lo mismo que el uso normal de la entrada estándar excepto por la etiqueta. Podríamos haber utilizado un *here-document* en el ejemplo anterior de >>, así (EOF, por «end of file» o «fin de archivo», es una etiqueta que se usa con frecuencia):

```
$ cat >> .mailrc << EOF
> alias fred frederick@longmachinename.longcompanyname.com
> EOF
$
```

Los *here-documents* están destinados a ser utilizados desde scripts de shell; te permiten especificar entrada «por lotes» para programas. Un uso común de los *here-documents* es con editores de texto simples como *ed(1)*. La Tarea 7-1 utiliza un *here-document* de esta manera.

Tarea 7-1

El comando `s file` en *mail(1)* guarda el mensaje actual en un *archivo*. Si el mensaje proviene de una red (como Internet), tiene varias líneas de encabezado antepuestas que proporcionan información sobre la ruta de la red. Necesitas esta información porque estás intentando resolver algunos problemas de enrutamiento de red. Escribe un script de shell que extraiga solo las líneas de encabezado del archivo.

Podemos usar *ed* para eliminar las líneas del cuerpo, dejando solo el encabezado. Para hacer esto, necesitamos saber algo sobre la sintaxis de los mensajes de correo, específicamente, que siempre hay una línea en blanco entre las líneas de encabezado y el texto del mensaje. El comando `ed /^$/,$d` hace el truco: significa «Eliminar desde la primera línea en blanco [2] hasta la última línea del archivo». También necesitamos los comandos `ed w` (escribir el archivo modificado) y `q` (salir). Aquí está el código que resuelve la tarea:

```
ed $1 << \EOF
/^$/,$d
w
q
EOF
```

Normalmente, el shell realiza sustitución de parámetros (variables), sustitución de comandos y sustitución aritmética en el texto de un *here-document*, lo que significa que puedes usar variables y comandos del shell para personalizar el texto. Esta evaluación se desactiva si alguna parte del delimitador está entre comillas, como se hizo en el ejemplo anterior. (Esto evita que el shell trate `$d` como una sustitución de variable).

A menudo, sin embargo, quieres que el shell realice sus evaluaciones: quizás el uso más común de los *here-documents* es proporcionar plantillas para generadores de formularios o texto de programas para generadores de programas. La Tarea 7-2 es una tarea sencilla para administradores de sistemas que muestra cómo funciona esto.

Tarea 7-2

Escribe un script que envíe un mensaje de correo a un conjunto de usuarios informando que se ha instalado una nueva versión de cierto programa en un directorio específico.

Puedes obtener una lista de todos los usuarios en el sistema de varias maneras; quizás la

[2] La línea tiene que estar completamente vacía; sin espacios ni TABs. Está bien: los encabezados de los mensajes de correo están separados de sus cuerpos exactamente por este tipo de línea en blanco.

más fácil es usar *cut* para extraer el primer campo de /etc/passwd, el archivo que contiene toda la información de la cuenta de usuario. Los campos en este archivo están separados por dos puntos (:).[3]

Dada una lista de usuarios, el siguiente código hace el truco:

```
pgmname=$1
for user in $(cut -f1 -d: /etc/passwd); do
  mail $user << EOF
  Dear $user,
  A new version of $pgmname has been installed in $(whence pgmname).
  Regards,
  Your friendly neighborhood sysadmin.
EOF
done
```

El shell sustituye los valores apropiados para el nombre del programa y su directorio.

El redireccionador `<<` tiene dos variaciones. Primero, puedes evitar que el shell realice la sustitución de parámetros, comandos y aritmética al rodear la *etiqueta* con comillas simples o dobles. (En realidad, es suficiente citar solo un carácter en la *etiqueta*.) Vimos esto en la solución para la Tarea 7-1.

La segunda variación es `<<-`, que elimina los TABs iniciales (pero no los espacios) del here-document y de la línea de etiqueta. Esto te permite sangrar el texto del here-document, haciendo que el script del shell sea más legible:

```
pgmname=$1
for user in $(cut -f1 -d: /etc/passwd); do
  mail $user <<- EOF
  Dear $user,
  A new version of $pgmname has been installed in $(whence pgmname).
  Regards,
  Your friendly neighborhood sysadmin.
EOF
done
```

Por supuesto, debes elegir tu *etiqueta* para que no aparezca como una línea de entrada real.

7.1.2. Here-Strings

Un giro común en la programación de shell es usar `print` para generar un texto que será procesado aún más por uno o más comandos:

[3]Hay algunos problemas posibles con esto; por ejemplo, `/etc/passwd` generalmente contiene información sobre «cuentas» que no están asociadas con personas, como `uucp`, `lp` y `daemon`. Ignoraremos tales problemas para el propósito de este ejemplo.

```
# empezar con un interrogatorio suave
print -r "$name, $rank, $serial_num" | interrogate -i mild
```

Esto podría reescribirse para usar un *here-document* que es ligeramente más eficiente, aunque no necesariamente más fácil de leer:

```
# empezar con un interrogatorio suave
interrogate -i mild << EOF
$name, $rank, $serial_num
EOF
```

A partir de *ksh93n*, [4] el shell de Korn proporciona una nueva forma de here-document, usando tres signos de menor que:

```
program <<< WORD
```

En esta forma, el texto de WORD (seguido de un salto de línea final) se convierte en la entrada del programa. Por ejemplo:

```
# empezar con un interrogatorio suave
interrogate -i mild <<< "$name, $rank, $serial_num"
```

Esta notación se originó por primera vez en la versión de Unix del shell `rc`, donde se llama «here string». Luego fue adoptada por el shell Z, *zsh* (ver Apéndice A), del cual el shell de Korn lo tomó prestado. Esta notación es simple, fácil de usar, eficiente y visualmente distinguible de los *here-documents* regulares.

7.1.3. Descriptores de ficheros

Los siguientes redireccionadores en la Tabla 7.1 dependen de la noción de un *descriptor de archivo*. Este es un concepto de E/S (Entrada/Salida) de bajo nivel en Unix que es vital entender al programar en C o C++. Aparece a nivel del shell cuando quieres hacer algo que no involucra la entrada estándar, la salida estándar y el error estándar. Puedes manejarte con algunos hechos básicos sobre ellos; para toda la información, consulta las entradas *open(2), creat(2), read(2), write(2), dup(2), dup2(2), fcntl(2)* y *close(2)* en el manual de Unix. (Como las entradas del manual están dirigidas al programador en C, su relación con los conceptos del shell no será necesariamente obvia).

Los descriptores de archivo son enteros que comienzan en 0 que indexan un array de información de archivos dentro de un proceso. Cuando un proceso se inicia, tiene tres descriptores de archivo abiertos. Estos corresponden a los tres estándares: entrada estándar

[4] Gracias a David Korn por proporcionarme acceso previo a la versión con esta característica. ADR.

(descriptor de archivo 0), salida estándar (1) y error estándar (2). Si un proceso abre archivos Unix para entrada o salida, se les asignan los siguientes descriptores de archivo disponibles, comenzando por 3.

De lejos, el uso más común de los descriptores de archivo con el shell de Korn es guardar el error estándar en un archivo. Por ejemplo, si deseas guardar los mensajes de error de un trabajo largo en un archivo para que no desplacen la pantalla, agrega `2> archivo` a tu comando. Si también deseas guardar la salida estándar, agrega `> archivo1 2> archivo2`.

Esto nos lleva a la Tarea 7-3.

Tarea 7-3

Quieres iniciar un trabajo largo en segundo plano (para liberar tu terminal) y guardar tanto la salida estándar como el error estándar en un solo archivo de registro. Escribe una función que haga esto.

Llamaremos a esta función `start`. El código es muy conciso:

```
function start {
  "$@" > logfile 2>&1 &
}
```

Esta línea ejecuta cualquier comando y parámetros que sigan a `start`. (El comando no puede contener tuberías ni redireccionadores de salida). Primero envía la salida estándar del comando a `logfile`.

Luego, el redireccionador `2>&1` dice: «Envía el error estándar (descriptor de archivo 2) al mismo lugar que la salida estándar (descriptor de archivo 1)». `2>&1` es en realidad una combinación de dos redireccionadores en la Tabla 7-1: `n> archivo` y `>&n`. Dado que la salida estándar se redirige a `logfile`, el error estándar también irá allí. El `&` final pone el trabajo en segundo plano para que recuperes tu indicador del shell.

Como una pequeña variación en este tema, podemos enviar tanto la salida estándar como el error estándar a una tubería en lugar de un archivo: `comando 2>&1 | ...` hace esto. (Por qué esto funciona se describe en breve). Aquí hay una función que envía tanto la salida estándar como el error estándar al archivo de registro (como se mencionó anteriormente) y al terminal:

```
function start {
  "$@" 2>&1 | tee logfile &
}
```

El comando *tee(1)* toma su entrada estándar y la copia en la salida estándar y en el archivo dado como argumento.

Estas funciones tienen una desventaja: debes permanecer conectado hasta que se complete el trabajo. Aunque siempre puedes escribir `jobs` (ver Capítulo 1) para verificar el progreso, no puedes irte de tu oficina por el día a menos que quieras arriesgarte a una violación de seguridad o desperdiciar electricidad. Veremos cómo resolver este problema en el Capítulo 8.

Los demás redireccionadores orientados a descriptores de archivos (por ejemplo, `<&n`) se usan generalmente para leer entrada (o escribir salida) desde (o hacia) más de un archivo al mismo tiempo. Veremos un ejemplo más adelante en este capítulo. De lo contrario, están destinados principalmente a programadores de sistemas, al igual que `<&-` (forzar el cierre de la entrada estándar) y `>&-` (forzar el cierre de la salida estándar), `<&n-` (mover el descriptor de archivo n a la entrada estándar) y `>&n-` (mover el descriptor de archivo n a la salida estándar).

Finalmente, solo debemos señalar que `0<` es lo mismo que `<`, y `1>` es lo mismo que `>`. (De hecho, 0 es el valor predeterminado para cualquier operador que comience con `<`, y 1 es el valor predeterminado para cualquier operador que comience con `>`).

Orden de redireccionamiento

El shell procesa las redirecciones de E/S en un orden específico. Una vez que entiendas cómo funciona esto, puedes aprovecharlo, especialmente para gestionar la disposición de la salida estándar y el error estándar.

Lo primero que hace el shell es configurar la entrada y salida estándar para las canalizaciones según lo indicado por el carácter |. Después de eso, procesa el cambio de descriptores de archivo individuales. Como acabamos de ver, el idioma más común que se beneficia de esto es enviar tanto la salida estándar como el error estándar por la misma canalización a un programa de paginación, como *more* o *less*. [5]

```
$ mycommand -h fred -w wilma 2>&1 | more
```

En este ejemplo, el shell primero establece la salida estándar de `mycommand` como la tubería a *more*. Luego redirige el error estándar (descriptor de archivo 2) para que sea igual a la salida estándar (descriptor de archivo 1), es decir, la tubería.

[5] *less* es un programa de paginación no estándar pero comúnmente disponible que tiene más funciones que *more*.

Cuando se trabaja solo con redireccionadores, se procesan de izquierda a derecha, según aparecen en la línea de comandos. Un ejemplo similar al siguiente ha estado en la página del manual del shell desde la versión original 7 de Bourne shell:

```
program > file1 2>&1          # Salida estándar y error estándar a file1
program 2>&1 > file1          # Error estándar al terminal y salida estándar a file1
```

En el primer caso, la salida estándar se envía a `file1` y luego el error estándar se envía al mismo lugar que la salida estándar, es decir, `file1`. En el segundo caso, el error estándar se envía al mismo lugar que la salida estándar, que sigue siendo el terminal. La salida estándar se redirige luego a `file1`, pero solo la salida estándar. Si comprendes esto, probablemente sepas todo lo que necesitas saber sobre los descriptores de archivo.

7.1.4. Nombres de archivos especiales

Normalmente, cuando proporcionas una ruta de acceso después de un redireccionador de E/S como `<` o `>`, el shell intenta abrir un archivo real con el nombre de archivo dado. Sin embargo, hay dos tipos de rutas de acceso donde el shell trata las rutas de acceso de manera especial.

El primer tipo de ruta de acceso es `/dev/fd/N`, donde *N* es el número de descriptor de archivo de un archivo que ya está abierto. Por ejemplo:

```
# supongamos que el descriptor de archivo 6 ya está abierto en un archivo
print 'algo significativo' > /dev/fd/6      # igual que 1>&6
```

Esto funciona incluso en sistemas que no tienen un directorio `/dev/fd`. Este tipo de ruta de acceso también se puede utilizar con los varios operadores de prueba de atributos de archivos del comando `[[...]]`.

El segundo tipo de ruta de acceso permite el acceso a servicios de Internet a través de los protocolos TCP o UDP. Las rutas de acceso son:

/dev/tcp/host/port Usando TCP, se conecta al host remoto en el *puerto* remoto. El host puede darse como una dirección IP en notación decimal con puntos (1.2.3.4) o como un nombre de host (www.oreilly.com). De manera similar, el puerto para el servicio deseado puede ser un nombre simbólico (típicamente encontrado en `/etc/services`) o un número de puerto numérico.[6]

/dev/udp/host/port Esto es lo mismo, pero usando UDP.

[6] La posibilidad de utilizar nombres de host se añadió en *ksh93f*; el uso de nombres de servicio se añadió en *ksh93m*.

Para usar estos archivos para E/S bidireccional, abre un nuevo descriptor de archivo usando el comando integrado exec (que se describe en el Capítulo 9), usando el operador «leer y escribir», `<>`. Luego utiliza `read -u` y `print -u` para leer y escribir en el nuevo descriptor de archivo. (El comando *read* y la opción `-u` para *read* y *print* se describen más adelante en este capítulo).

El siguiente ejemplo, cortesía de David Korn, muestra cómo hacer esto. Implementa el programa *whois(1)*, que proporciona información sobre el registro de nombres de dominio en Internet:

```
host=rs.internic.net
port=43
exec 3<> /dev/tcp/$host/$port
print -u3 -f "%s\r\n" "$@"
cat <&3
```

Usando el comando integrado `exec` (ver Capítulo 9), este programa utiliza el operador «leer y escribir», `<>`, para abrir una conexión bidireccional al host *rs.internic.net* en el puerto TCP 43, que proporciona el servicio *whois*. (El script también podría haber usado `port=whois`). Luego utiliza el comando `print` para enviar las cadenas de argumentos al servidor `whois`. Finalmente, lee el resultado devuelto usando *cat*. Aquí hay una ejecución de muestra:

```
$ whois.ksh kornshell.com
Whois Server Version 1.3

Domain names in the .com, .net, and .org domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: KORNSHELL.COM
   Registrar: NETWORK SOLUTIONS, INC.
   Whois Server: whois.networksolutions.com
   Referral URL: http://www.networksolutions.com
   Name Server: NS4.PAIR.COM
   Name Server: NS0.NS0.COM
   Updated Date: 02-dec-2001

>>> Last update of whois database: Sun, 10 Feb 2002 05:19:14 EST <<<

The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains and
Registrars.
```

La programación de redes está más allá del alcance de este libro. Pero para la mayoría de las cosas, probablemente querrás usar conexiones TCP en lugar de conexiones UDP si escribes algún programa de red en *ksh*.

7.2. Cadenas de E/S

Ahora volveremos al nivel de E/S de cadenas y examinaremos las declaraciones `print`, `printf` y `read`, que proporcionan al shell capacidades de E/S más análogas a los lenguajes de programación convencionales.

7.2.1. print

Como hemos visto innumerables veces en este libro, `print` simplemente imprime sus argumentos en la salida estándar. Deberías usarlo en lugar del comando echo, cuya funcionalidad difiere de un sistema a otro.[7] (La versión incorporada de *echo* en el shell de Korn emula lo que hace la versión estándar de `echo` del sistema). Ahora exploraremos el comando `print` con más detalle.

Secuencias de escape para print

`print` acepta varias opciones, así como varias secuencias de escape que comienzan con una barra invertida. (Debes usar una doble barra invertida si no rodeas la cadena que las contiene con comillas; de lo contrario, la propia shell «roba» una barra invertida antes de pasar los argumentos a `print`). Estas son similares a las secuencias de escape reconocidas por `echo` y el lenguaje C; se enumeran en la Tabla 7.2.

Estas secuencias muestran un comportamiento bastante predecible, excepto `\f`. En algunas pantallas, provoca un borrado de pantalla, mientras que en otras provoca un avance de línea. Expulsa la página en la mayoría de las impresoras. `\v` está algo obsoleto; generalmente provoca un avance de línea.

Tabla 7.2: Secuencias de escape de print

Secuencia	**Caracter impreso**
\a	ALERT o CTRL-G
\b	BACKSPACE o CTRL-H
\c	Omitir la nueva línea final y dejar de procesar la cadena
\E	ESCAPE o CTRL-[
\f	FORMFEED o CTRL-L

[7] Específicamente, hay una diferencia entre las versiones System V y BSD. Este último acepta opciones similares a las de `print`, mientras que el primero acepta secuencias de escape al estilo del lenguaje C.

Secuencia	Caracter impreso
\n	Nueva línea (no al final del comando) o CTRL-J
\r	ENTER (RETURN) o CTRL-M
\t	TAB o CTRL-I
\v	VERTICAL TAB o CTRL-K
\0n	Carácter ASCII con valor octal (base-8) *n*, donde *n* tiene de 1 a 3 dígitos. A diferencia de C, C++ y muchos otros lenguajes, se requiere el 0 inicial.
\\	Barra diagonal inversa simple

La secuencia `\0n` es aún más dependiente del dispositivo y se puede utilizar para E/S compleja, como el control del cursor y caracteres gráficos especiales.

Opciones para print

`print` también acepta algunas opciones con guión; ya hemos visto `-n` para omitir el salto de línea final. Las opciones se enumeran en la Tabla 7.3.

Tabla 7.3: Opciones de print

Opción	Función
-e	Procesar secuencias de escape en los argumentos (este es el valor por defecto).
-f *format*	Imprime como si fuera mediante *printf* con el formato dado (véase la sección siguiente).
-n	Omite la nueva línea final (igual que la secuencia de escape \c).
-p	Imprime en la tubería a la coroutine; ver Capítulo 8.
-r	Raw; ignora las secuencias de escape listadas arriba.
-R	Como -r, pero además ignora cualquier otra opción excepto -n.
-s	Imprime en un archivo de historial de comandos (véase el Capítulo 2).
-un	Imprime en el descriptor de archivo n.

Observa que algunas de estas son redundantes: `print -n` es lo mismo que `print` con `\c` al final de una línea; `print -un ...` es equivalente a `print ... >&n` (aunque la primera es ligeramente más eficiente).

Sin embargo, `print -s` no es lo mismo que `print ... >> $HISTFILE`. El último comando deja temporalmente inoperables los modos de edición *vi* y *emacs*; debes usar `print -s` si deseas imprimir en tu archivo de historial.

Imprimir en tu archivo de historial es útil si deseas editar algo que el shell expande cuando procesa una línea de comandos, por ejemplo, una variable de entorno compleja como PATH.

Si ingresas el comando `print -s PATH=$PATH`, presionas ENTER y luego presionas CTRL-P en modo emacs (o ESC k en modo *vi*), verás algo como esto:

```
$ PATH=/bin:/usr/bin:/etc:/usr/ucb:/usr/local/bin:/home/billr/bin
```

Es decir, el shell expande la variable (y cualquier otra cosa, como sustituciones de comandos, comodines, etc.) antes de escribir la línea en el archivo de historial. Tu cursor estará al final de la línea (o al principio de la línea en modo *vi*), y podrás editar tu PATH sin tener que volver a escribir todo.

7.2.2. printf

Si necesitas producir informes formateados, el comando *print* del shell se puede combinar con atributos de formato para variables y así producir datos de salida que se alinean razonablemente. Pero solo puedes hacer tanto con estas facilidades.

La rutina de biblioteca *printf(3)* del lenguaje C proporciona potentes instalaciones de formato para un control total de la salida. Es tan útil que muchos otros lenguajes de programación derivados de Unix, como *awk* y *perl*, admiten instalaciones similares o idénticas. Principalmente debido a que el comportamiento de *echo* en diferentes sistemas Unix no se podía reconciliar y reconociendo la utilidad de *printf*, el estándar de shell POSIX exige un comando *printf* a nivel de shell que proporcione la misma funcionalidad que la rutina de biblioteca *printf(3)*. Esta sección describe cómo funciona el comando *printf* y examina capacidades adicionales únicas de la versión de *printf* del shell de Korn.

El comando *printf* puede emitir una cadena simple al igual que el comando *print*.

```
printf "¡Hola, mundo!\n"
```

La principal diferencia que notarás de inmediato es que, a diferencia de *print*, *printf* no suministra automáticamente un salto de línea. Debes especificarlo explícitamente como `\n`.

La sintaxis completa del comando *printf* tiene dos partes:

```
printf cadena-de-formato [argumentos ...]
```

La primera parte es una cadena que describe las especificaciones de formato; esto se suministra mejor como una constante de cadena entre comillas. La segunda parte es una lista de argumentos, como una lista de cadenas o valores de variables, que corresponden a las especificaciones de formato. (Si hay más argumentos que especificaciones de formato, *ksh* recorre las especificaciones de formato en la cadena de formato, reutilizándolas en orden,

hasta que se complete). Una especificación de formato se precede de un signo de porcentaje (%), y el especificador es uno de los caracteres que se describen en breve. Dos de los principales especificadores de formato son `%s` para cadenas y `%d` para enteros decimales.

La cadena de formato combina texto para ser emitido literalmente con especificaciones que describen cómo formatear los argumentos subsiguientes en la línea de comandos de *printf*. Por ejemplo:

```
$ printf "¡Hola, %s!\n" Mundo
¡Hola, Mundo!
```

Dado que el comando *printf* está incorporado, no estás limitado a números absolutos:

```
$ printf "La respuesta es %d.\n" 12+10+20
La respuesta es 42.
```

Los especificadores permitidos se muestran en la Tabla 7.4.

Tabla 7.4: Especificadores de formato utilizados en *printf*

Especificador	**Descripción**
%c	Carácter ASCII (imprime el primer carácter del argumento correspondiente)
%d	Número entero decimal
%i	Número entero decimal
%e	Formato de coma flotante ([-]d.precisióne[+-]dd) (véase el texto siguiente para el significado de la precisión)
%E	Formato de coma flotante ([-]d.precisiónE[+-]dd)
%f	Formato de coma flotante ([-]ddd.precisión)
%g	Conversión %e o %f, la que sea más corta, sin ceros al final
%G	Conversión %E o %f, la que sea más corta, sin ceros al final.
%o	Valor octal sin signo
%s	Cadena (string)
%u	Valor decimal sin signo
%x	Número hexadecimal sin signo. Usa a-f para 10 al 15
%X	Número hexadecimal sin signo. Usa A-F para 10 al 15
%%	% literal

El comando *printf* se puede utilizar para especificar el ancho y la alineación de los campos de salida. Una expresión de formato puede tener tres modificadores opcionales que siguen a % y preceden al especificador de formato:

```
%flags width.precision format-specifier
```

El *ancho* del campo de salida es un valor numérico. Cuando especificas un ancho de campo, el contenido del campo se justifica a la derecha de forma predeterminada. Debes especificar una bandera de «-» para obtener justificación a la izquierda. (El resto de las banderas se

discuten en breve). Por lo tanto, « %-20s» imprime una cadena justificada a la izquierda en un campo de 20 caracteres de ancho. Si la cadena tiene menos de 20 caracteres, el campo se rellena con espacios en blanco para completar. En los siguientes ejemplos, se imprime un | para indicar el ancho real del campo. El primer ejemplo justifica el texto a la derecha:

```
printf "|%10s|\n" hello
```

Produce:

```
|     hello|
```

El siguiente ejemplo justifica el texto a la izquierda:

```
printf "|%-10s|\n" hello
```

Produce:

```
|hello     |
```

El modificador de *precisión*, utilizado para valores decimales o de punto flotante, controla la cantidad de dígitos que aparecen en el resultado. Para valores de cadena, controla el número máximo de caracteres de la cadena que se imprimirán.

Puedes especificar tanto el *ancho* como la *precisión* dinámicamente, mediante valores en la lista de argumentos de *printf*. Haces esto especificando asteriscos, en lugar de valores literales.

```
$ myvar=42.123456
$ printf "|%*.*G|\n" 5 6 $myvar
|42.1235|
```

En este ejemplo, el ancho es 5, la precisión es 6 y el valor a imprimir proviene del valor de `myvar`.

La *precisión* es opcional. Su significado exacto varía según la letra de control, como se muestra en la Tabla 7.5:

Tabla 7.5: Significado de la precisión

Conversión	**Significado de la presición**
%d, %i, %o, %u, %x, %X	El número mínimo de dígitos a imprimir. Cuando el valor tiene menos dígitos, se rellena con ceros a la izquierda. La precisión por defecto es 1.

Conversión	Significado de la presición
%e, %E	El número mínimo de dígitos a imprimir. Cuando el valor tiene menos dígitos, se rellena con ceros después del punto decimal. La precisión por defecto es 10. Una precisión de 0 inhibe la impresión del punto decimal.
%f	El número de dígitos a la derecha del punto decimal.
%g, %G	El número máximo de dígitos significativos.
%s	El número máximo de caracteres a imprimir.

Finalmente, una o más *banderas* pueden preceder al ancho del campo y la precisión. Ya hemos visto la bandera «-» para la justificación a la izquierda. El resto de las banderas se muestran en la Tabla 7.6.

Tabla 7.6: Banderas para printf

Carácter	Descripción
-	Justifica a la izquierda el valor formateado dentro del campo.
space	Anteponga un espacio a los valores positivos y un signo menos a los negativos.
+	Anteponga siempre un signo a los valores numéricos, aunque el valor sea positivo.
#	Utilice una forma alternativa: %o tiene un 0 precedente; %x y %X están prefijados con 0x y 0X, respectivamente; %e, %E y %f siempre tienen un punto decimal en el resultado; y %g y %G no tienen ceros finales eliminados.
0	Rellenar la salida con ceros, no con espacios. Esto sólo ocurre cuando la anchura del campo es mayor que el resultado convertido. En el lenguaje C, esta bandera se aplica a todos los formatos de salida, incluso a los no numéricos. Para *ksh*, sólo se aplica a los formatos numéricos.

Si *printf* no puede realizar una conversión de formato, devuelve un estado de salida no nulo.

Similar a *print*, el comando *printf* incorporado interpreta secuencias de escape dentro de la cadena de formato. Sin embargo, *printf* acepta un rango más amplio de secuencias de escape; son las mismas que para la cadena `$'...'`. Estas secuencias se enumeran más adelante en la Tabla 7.9.

7.2.3. Especificadores adicionales de printf en el shell de Korn

Además de los especificadores estándar recién descritos, el shell de Korn acepta varios especificadores adicionales. Estos proporcionan funciones útiles a expensas de la no portabilidad a otras versiones del comando *printf*.

$b Cuando se utiliza en lugar de %s, expande las secuencias de escape de estilo de impresión en la cadena del argumento. Por ejemplo:

```
$ printf "%s\n" 'hola\nmundo'
hola\nmundo
$ printf "%b\n" 'hola\nmundo'
hola
mundo
```

%H Cuando se utiliza en lugar de %s, muestra los caracteres especiales HTML y XML como sus correspondientes nombres de entidad. Por ejemplo:

```
$ printf "%s\n" "Aquí hay caracteres < y > reales"
Aquí hay caracteres < y > reales
$ printf "%H\n" "Aquí hay caracteres < y > reales"
Aquí hay caracteres &lt; y &gt; reales
```

Curiosamente, los espacios se convierten en , el carácter de espacio literal irrompible de HTML y XML.

%n Esto es un préstamo de ISO C. Coloca el número de caracteres escritos hasta el momento en la variable dada. Esto es posible ya que *printf* está incorporado en el shell.

```
$ printf "hola, mundo\n%n" msglen
hola, mundo
$ print $msglen
13
```

%P Cuando se utiliza en lugar de %s, traduce la expresión regular extendida de estilo *egrep* a un patrón de shell de Korn equivalente. Por ejemplo:

```
$ printf "%P\n" '(.*\.o|.*\.obj|core)+'
*+(*\.o|*\.obj|core)*
```

%q Cuando se utiliza en lugar de %s, imprime el argumento de cadena entre comillas de forma que pueda reutilizarse posteriormente dentro de un script de shell. Por ejemplo:

```
$ printf "print %q\n" "una cadena con ' y \" en ella"
print $'una cadena con \' y " en ella'
```

(La notación `$'...'` se explica en la Sección 7.3.3.1, más adelante en este capítulo).

%R Va a la inversa de %P, traduciendo patrones en expresiones regulares extendidas. Por ejemplo:

```
$ printf "%R\n" '+(*.o|*.c)'
^(.*\.o|.*\.c)+$
```

%(date format)T El formato de *date* es una cadena de comandos de fecha similar a la de *date(1)*. El argumento es una cadena que representa una fecha y una hora. *ksh* convierte la cadena de fecha dada en la hora que representa y luego la reformatea según el formato de *date(1)* que usted suministre. *ksh* acepta una amplia variedad de formatos de fecha y hora. Por ejemplo:

```
$ date
Wed Jan 30 15:46:01 IST 2002
$ printf "%(Ahora es %m/%d/%Y %H:%M:%S)T\n" "$(date)"
Ahora es 30/01/2002 15:46:07
```

Los sistemas Unix mantienen el tiempo en «segundos desde Epoch». *Epoch* es la medianoche del 1 de enero de 1970, UTC. Si dispone de un valor de tiempo en este formato, puede utilizarlo con el especificador de conversión %T precediéndolo de un carácter #, así:

```
$ printf "%(It is now %m/%d/%Y %H:%M:%S)T\n" '#1012398411'
Ahora es 30/01/2002 15:46:51
```

%z Imprime un byte cuyo valor es cero.

Por último, para el formato %d, después de la precisión puede suministrar un punto adicional y un número que indique la base de salida:

```
$ printf '42 es %.3.5d en base 5\n' 42
42 es 132 en base 5
```

7.2.4. read

La otra cara de las facilidades de E/S de cadena del shell es el comando *read*, que te permite leer valores en variables del shell. La sintaxis básica es:

```
read var1 var2 ...
```

Hay algunas opciones, que cubrimos en la Sección 7.2.3.5, más adelante en este capítulo. Esta instrucción toma una línea de la entrada estándar y la descompone en palabras delimitadas por cualquiera de los caracteres en el valor de la variable `IFS` (ver Capítulo 4; estos suelen ser un espacio, un TAB y un salto de línea). Las palabras se asignan a las variables *var1, var2,* etc. Por ejemplo:

```
$ read fred bob
dave pete
$ print "$fred"
dave

$ print "$bob"
pete
```

Si hay más palabras que variables, las palabras excedentes se asignan a la última variable. Si omites las variables por completo, toda la línea de entrada se asigna a la variable `REPLY`.

Puede que hayas identificado esto como el ingrediente que falta en las capacidades de programación de scripts del shell que hemos visto hasta ahora. Se asemeja a las declaraciones de entrada en lenguajes convencionales, como su homónimo en Pascal. Entonces, ¿por qué esperamos tanto para introducirlo?

En realidad, *read* es una especie de escape de la filosofía tradicional de programación de scripts del shell, que dicta que la unidad de datos más importante para procesar es un *archivo de texto* y que se deben utilizar utilidades de Unix como *cut, grep, sort,* etc., como bloques de construcción para escribir programas.

read, por otro lado, implica procesamiento línea por línea. Podrías usarlo para escribir un script del shell que haga lo que normalmente haría una tubería de utilidades, pero tal script inevitablemente se vería así:

```
while (read a line) do
    procesar la línea
    imprimir la línea procesada
end
```

Este tipo de script suele ser mucho más lento que una tubería; además, tiene la misma forma que un programa que alguien podría escribir en C (o un lenguaje similar) que hace lo mismo mucho, *mucho* más rápido. En otras palabras, si vas a escribirlo de esta manera línea por línea, no tiene sentido escribir un script del shell. (Los autores han pasado años sin escribir un script con *read* en él).

Lectura de líneas de ficheros

Sin embargo, los scripts de shell con *read* son útiles para ciertos tipos de tareas. Uno de ellos es cuando estás leyendo datos de un archivo lo suficientemente pequeño como para que la eficiencia no sea una preocupación (digamos unas pocas cientos de líneas o menos), y realmente es necesario colocar fragmentos de entrada en variables del shell.

Una tarea que ya hemos visto se ajusta a esta descripción: la Tarea 5-4, el script que un administrador del sistema podría usar para establecer la variable de entorno TERM de un usuario según la línea de terminal que esté utilizando. El código en el Capítulo 5 utilizó una declaración `case` para seleccionar el valor correcto para `TERM`.

Este código presumiblemente residiría en `/etc/profile`, el archivo de inicialización del sistema que el shell de Korn ejecuta antes de ejecutar el *.profile* de un usuario. Si las terminales en el sistema cambian con el tiempo, como seguramente lo harán, entonces el código tendría que cambiarse. Sería mejor almacenar la información en un archivo y cambiar solo el archivo en su lugar.

Supongamos que colocamos la información en un archivo cuyo formato es típico de los archivos de «configuración del sistema» de Unix: cada línea contiene un nombre de dispositivo, un TAB y un valor de TERM. Si el archivo, que llamaremos `/etc/terms`, contenía los mismos datos que la declaración `case` en el Capítulo 5, se vería así:

```
console  s531
tty01    gl35a
tty03    gl35a
tty04    gl35a
tty07    t2000
tty08    s531
```

Podemos usar *read* para obtener los datos de este archivo, pero primero necesitamos saber cómo probar la condición de fin de archivo. Simple: el estado de salida de *read* es 1 (es decir, diferente de cero) cuando no hay nada que leer. Esto nos lleva a un bucle `while` limpio:

```
TERM=vt99 # asumir esto como un valor predeterminado
line=$(tty)
while read dev termtype; do
    if [[ $dev == $line ]]; then
        TERM=$termtype
        export TERM
        print "TERM se estableció en $TERM."
        break
    fi
done
```

El bucle `while` lee cada línea de la entrada en las variables `dev` y `termtype`. En cada paso a través del bucle, el `if` busca una coincidencia entre `$dev` y la tty del usuario (`$line`, obtenida por la sustitución de comandos desde el comando *tty*). Si se encuentra una coincidencia, se establece `TERM` y se exporta, se imprime un mensaje y el bucle sale; de lo contrario, `TERM` permanece en el valor predeterminado de `vt99`.

Aún no hemos terminado: ¡este código lee desde la entrada estándar, no desde `/etc/terms`! Necesitamos saber cómo redirigir la entrada a múltiples comandos. Hay varias maneras de hacer esto.

Redirección de E/S y comandos múltiples

Una forma de resolver el problema es con un subproceso, como veremos en el próximo capítulo. Esto implica crear un proceso separado para realizar la lectura. Sin embargo, generalmente es más eficiente hacerlo en el mismo proceso; el shell de Korn nos ofrece tres formas de hacer esto.

La primera, que ya hemos visto, es con una función:

```
function findterm {
    TERM=vt99                       # asumir esto como un valor predeterminado
    line=$(tty)
    while read dev termtype; do
        if [[ $dev == $line ]]; then
            TERM=$termtype
            export TERM
            print "TERM se estableció en $TERM."
            break
        fi
    done
}
findterm < /etc/terms
```

Una función actúa como un script en el sentido de que tiene su propio conjunto de descriptores de E/S estándar, que se pueden redirigir en la línea de código que llama a la función. En otras palabras, puedes pensar en este código como si *findterm* fuera un script y escribieras `findterm < /etc/terms` en la línea de comandos. La declaración *read* toma la entrada de `/etc/terms` línea por línea, y la función se ejecuta correctamente.

La segunda forma es colocando el redireccionador de E/S al final del bucle, de esta manera:

```
TERM=vt99                           # asumir esto como un valor predeterminado
line=$(tty)
while read dev termtype; do
    if [[ $dev == $line ]]; then
        TERM=$termtype
        export TERM
        print "TERM se estableció en $TERM."
        break
    fi
done < /etc/terms
```

Puedes usar esta técnica con cualquier construcción de control de flujo, incluyendo `if...fi`,

`case...esac`, `for...done`, `select...done` y `until...done`. Esto tiene sentido porque todas estas son *declaraciones compuestas* que el shell trata como comandos únicos para estos propósitos. Esta técnica funciona bien, ya que el comando *read* lee una línea a la vez, siempre y cuando toda la entrada se realice dentro de la declaración compuesta.

Colocar el redireccionador de E/S al final es especialmente importante para que los bucles funcionen correctamente. Supongamos que colocas el redireccionador después del comando 'read', así:

```
while read dev termtype < /etc/terms
do
    ...
done
```

En este caso, el shell vuelve a abrir `/etc/terms` cada vez que entra en el bucle, leyendo la primera línea una y otra vez. Esto crea efectivamente un bucle infinito, algo que probablemente no deseas.

Bloques de código

En ocasiones, es posible que desees redirigir la entrada/salida hacia o desde un grupo arbitrario de comandos sin crear un proceso separado. Para hacer eso, necesitas utilizar una construcción que aún no hemos visto. Si rodeas un código con { y },[8] el código se comportará como una función sin nombre. Este es otro tipo de declaración compuesta. De acuerdo con el concepto equivalente en el lenguaje C, llamaremos a esto un bloque de código.[9]

¿Para qué sirve un bloque? En este caso, significa que el código dentro de las llaves ({ }) tomará descriptores de entrada/salida estándar tal como lo describimos para las funciones. Esta construcción también es apropiada para el ejemplo actual porque el código solo necesita ser llamado una vez, y el script completo no es lo suficientemente grande como para justificar descomponerlo en funciones. Así es como usamos un bloque en el ejemplo:

```
{
    TERM=vt99                                   # asumir esto como un valor predeterminado
    line=$(tty)
    while read dev termtype; do
        if [[ $dev == $line ]]; then
            TERM=$termtype
            export TERM
```

[8] Por razones sintácticas oscuras e históricas, las llaves son *palabras clave* del shell. En la práctica, esto significa que el cierre de } debe ir precedido por un salto de línea o un punto y coma. ¡Caveat emptor!

[9] Los programadores de LISP pueden preferir pensar en esto como una *función anónima* o *función lambda*.

```
            print "TERM se estableció en $TERM."
            break
        fi
    done
} < /etc/terms
```

Para ayudarte a entender cómo funciona esto, piensa en las llaves y el código dentro de ellas como si fueran un solo comando, es decir:

```
{ TERM=vt99; line=$(tty); while ... ; } < /etc/terms
```

Los archivos de configuración para tareas de administración del sistema como esta son bastante comunes; un ejemplo prominente es `/etc/hosts`, que enumera las máquinas que son accesibles en una red TCP/IP. Podemos hacer que `/etc/terms` sea más parecido a estos archivos estándar al permitir líneas de comentarios en el archivo que comiencen con #, al igual que en los scripts de shell. De esta manera, `/etc/terms` puede verse así:

```
#
# La consola del sistema es una Shande 531s
console s531
#
# La línea del Prof. Subramaniam tiene un Givalt GL35a
tty01 gl35a
...
```

Podemos manejar las líneas de comentarios de dos maneras. Primero, podríamos modificar el bucle `while` para que ignore las líneas que comienzan con #. Aprovecharíamos el hecho de que los operadores de igualdad y desigualdad (== y !=) dentro de [[...]] realizan coincidencia de patrones, no solo pruebas de igualdad:

```
if [[ $dev != \#* && $dev == $line ]]; then
...
```

El patrón es #*, que coincide con cualquier cadena que comience con #. Debemos preceder # con una barra invertida para que el shell no trate el resto de la línea como un comentario. Además, recuerda del Capítulo 5 que && combina las dos condiciones de manera que ambas deben ser verdaderas para que toda la condición sea verdadera.

Esto ciertamente funcionaría, pero la forma habitual de filtrar las líneas de comentarios es usar una canalización con *grep*. Le proporcionamos a *grep* la expresión regular ^[^#], que coincide con cualquier cosa excepto líneas que comienzan con #. Luego, cambiamos la llamada al bloque para que lea desde la salida de la canalización en lugar de leer directamente desde el archivo.[10]

[10] Desafortunadamente, usar *read* con entrada desde una tubería a menudo es muy ineficiente, debido a problemas en el diseño del shell que no son relevantes aquí.

```
grep "^[^#]" /etc/terms | {
    TERM=vt99
    ...
}
```

También podemos usar *read* para mejorar nuestra solución a la Tarea 6-3, en la que emulamos la salida en varias columnas de *ls*. En la solución del capítulo anterior, asumimos, para simplificar, que los nombres de archivo están limitados a 14 caracteres y usamos 14 como ancho de columna fijo. Mejoraremos la solución para que permita *cualquier* longitud de nombre de archivo (como en las versiones modernas de Unix) y utilice la longitud del nombre de archivo más largo (más 2) como el ancho de la columna.

Para mostrar la lista de archivos en formato de varias columnas, necesitamos leer la salida de *ls* dos veces. En el primer paso, encontramos el nombre de archivo más largo y lo usamos para establecer el número de columnas y su ancho; el segundo paso realiza la salida real. Aquí hay un bloque de código para el primer paso:

```
ls "$@" | {
    let width=0
    while read fname; do
        if (( ${#fname} > $width )); then
            let width=${#fname}
        fi
    done
    let "width += 2"
    let numcols="int(${COLUMNS:-80} / $width)"
}
```

Este código parece un ejercicio de una clase de programación del primer semestre. El bucle `while` recorre la entrada en busca de archivos con nombres más largos que el más largo encontrado hasta ahora; si se encuentra uno más largo, su longitud se guarda como la nueva longitud más larga.

Después de que el bucle termina, agregamos 2 al ancho para permitir espacio entre columnas. Luego dividimos el ancho del terminal por el ancho de la columna para obtener el número de columnas. Como el shell realiza la división en punto flotante, el resultado se pasa a la función *int* para producir un resultado final entero. Recuerda del Capítulo 3 que la variable integrada `COLUMNS` a menudo contiene el ancho de visualización; la construcción `${COLUMNS:-80}` da un valor predeterminado de 80 si esta variable no está configurada.

Los resultados del bloque son las variables `width` y `numcols`. Estas son variables globales, por lo que son accesibles por el resto del código dentro de nuestro script (eventual). En

particular, las necesitamos en nuestra segunda pasada por los nombres de archivo. El código para esto se asemeja al código de nuestra solución original; todo lo que necesitamos hacer es reemplazar el ancho de columna fijo y el número de columnas con las variables:

```
set -A filenames $(ls "$@")
typeset -L$width fname
let count=0

while (( $count < ${#filenames[*]} )); do
    fname=${filenames[$count]}
    print "$fname \c"
    let count++
    if [[ $((count % numcols)) == 0 ]]; then
        print # produce una nueva línea
    fi
done
if (( count % numcols != 0 )); then
    print
fi
```

El script completo consiste en ambas piezas de código. Como otro «ejercicio para el lector», considera cómo podrías reorganizar el código para invocar el comando *ls* solo una vez. (Pista: usa al menos un bucle aritmético `for`).

Lectura de la entrada del usuario

El otro tipo de tarea para la cual *read* es adecuado es solicitar la entrada del usuario. Piénsalo: apenas hemos visto scripts de este tipo hasta ahora en este libro. De hecho, los únicos fueron las soluciones modificadas para la Tarea 5-4, que involucraba `select`.

Como probablemente habrás deducido, *read* se puede utilizar para obtener la entrada del usuario en variables del shell. Podemos usar *print* para solicitar al usuario, así:

```
print -n '¿terminal? '
read TERM
print "TERM es $TERM"
```

Así es como se ve cuando se ejecuta:

```
¿terminal? vt99
TERM es vt99
```

Sin embargo, para que las solicitudes no se pierdan en una canalización, la convención del shell dicta que las solicitudes deben ir a la salida de error estándar, no a la salida estándar. (Recuerda que `select` solicita a la salida de error estándar). Podríamos simplemente usar el descriptor de archivo 2 con el redireccionador de salida que vimos anteriormente en este capítulo:

```
print -n '¿terminal? ' >&2
read TERM
print "TERM es $TERM"
```

El shell proporciona una mejor manera de hacer lo mismo: si sigues el primer nombre de variable en una declaración *read* con un signo de interrogación (?) y una cadena, el shell utiliza esa cadena como un indicador en la salida de error estándar. En otras palabras:

```
read TERM?'¿terminal? '
print "TERM es $TERM"
```

hace lo mismo que lo anterior. La forma del shell es mejor por las siguientes razones. Primero, esto se ve un poco mejor; en segundo lugar, el shell sabe que no debe generar el indicador si la entrada se redirige desde un archivo; y finalmente, este esquema te permite usar el modo *vi* o *emacs* en tu línea de entrada.

Ampliaremos este ejemplo simple mostrando cómo se haría la Tarea 5-4 si no existiera `select`. Compara esto con el código en el Capítulo 6:

```
set -A termnames gl35a t2000 s531 vt99
print 'Selecciona tu tipo de terminal:'
while true; do
{
    print '1) gl35a'
    print '2) t2000'
    print '3) s531'
    print '4) vt99'
} >&2
read REPLY?'¿terminal? '

if (( REPLY >= 1 && REPLY <= 4 )); then
    TERM=${termnames[REPLY-1]}
    print "TERM es $TERM"
    export TERM
    break
fi
done
```

El bucle `while` es necesario para que el código se repita si el usuario hace una elección no válida.

Esto es aproximadamente el doble de líneas de código que la primera solución en el Capítulo 5, ¡pero exactamente igual que la versión posterior, más amigable para el usuario! Esto muestra que `select` te ahorra código solo si no te importa usar las mismas cadenas para mostrar las opciones de tu menú que usas dentro de tu script.

Sin embargo, `select` tiene otras ventajas, como la capacidad de construir menús de varias columnas si hay muchas opciones y un mejor manejo de la entrada de usuario vacía.

Opciones para *read*

read acepta un conjunto de opciones que son similares a las de *print*. La Tabla 7.7 las enumera.

Tabla 7.7: Opciones de `read`

Opción	Función
-A	Lee palabras en una matriz indexada, empezando por el índice 0. Despliega primero todos los elementos de la matriz.
-d *delimitador*	Leer hasta el carácter delimitador, en lugar del carácter por defecto, que es una nueva línea.
-n *número*	Leer como máximo el número de bytes[11]
-p	Lectura de la tubería a la coroutine; ver Capítulo 8.
-r	Crudo; no utilice \ como carácter de continuación de línea.
-s	Guardar la entrada en el archivo de historial de comandos; véase el Capítulo 1.
-t *n segundos*	Espera hasta *n segundos* a que llegue la entrada. Si transcurren *n segundos*, devuelve un estado de salida de fallo.
-u*n*	Lectura del descriptor de archivo n.

Tener que escribir `read word[0] word[1] word[2] ...` para leer palabras en una matriz es tedioso. También propenso a errores; si el usuario escribe más palabras de las variables de la matriz proporcionadas, las palabras restantes se asignan todas a la última variable de la matriz. La opción `-A` resuelve esto, leyendo cada palabra una a la vez en las entradas correspondientes de la matriz nombrada.

La opción `-d` te permite leer hasta algún otro carácter que no sea una nueva línea. En términos prácticos, probablemente nunca necesitarás hacer esto, pero el shell quiere hacerlo posible por si alguna vez lo necesitas.

De manera similar, la opción `-n` te libera del modo predeterminado de *read* que consume la entrada línea por línea; te permite leer un número fijo de bytes. Esto es muy útil si estás procesando datos heredados de ancho fijo, aunque esto no es muy común en sistemas Unix.

read te permite ingresar líneas que son más largas que el ancho de tu dispositivo de visualización mediante el uso de la barra invertida (\) como un carácter de continuación, al igual que en los scripts de shell. La opción `-r` anula esto, en caso de que tu script lea desde un archivo que pueda contener líneas que terminan en barras invertidas.

[11] Esta opción fue añadida en *ksh93e*.

`read -r` también preserva cualquier otra secuencia de escape que la entrada pueda contener. Por ejemplo, si el archivo *fred* contiene esta línea:

```
A line with a\n escape sequence
```

`read -r fredline` incluirá la barra invertida en la variable `fredline`, mientras que sin `-r`, *read* «se comerá» la barra invertida. Como resultado:

```
$ read -r fredline < fred
$ print "$fredline"
A line with a
escape sequence
$
```

Aquí, *print* interpretó la secuencia de escape `\n` y la convirtió en un salto de línea. Sin embargo:

```
$ read fredline < fred
$ print "$fredline"
A line with an
escape sequence
$
```

La opción `-s` te ayuda si estás escribiendo un script altamente interactivo y deseas proporcionar la misma capacidad de historial de comandos que la propia shell tiene. Por ejemplo, supongamos que estás escribiendo una nueva versión de *mail* como un script de shell. Tu bucle básico de comandos podría verse así:

```
while read -s cmd; do
  # procesar el comando
done
```

El uso de `read -s` permite al usuario recuperar comandos anteriores para tu programa con el comando CTRL-P en modo *emacs* o el comando ESC k en modo *vi*. El depurador *kshdb* en el Capítulo 9 utiliza esta función.

La opción `-t` es bastante útil. Te permite recuperarte en caso de que tu usuario se haya «ido a almorzar», pero tu script tiene mejores cosas que hacer que esperar por la entrada. Le indicas cuántos segundos estás dispuesto a esperar antes de decidir que al usuario simplemente ya no le importa:

```
print -n "OK, Sr. $prisoner, ingrese su nombre, rango y número de serie: "
# esperar dos horas, no más
if read -t $((60 * 60 * 2)) name rank serial
then
  # procesar la información
  ...
else
```

```
  # el prisionero está siendo silencioso
  print '¿El tratamiento silencioso, eh? Solo espera.'
  call_evil_colonel -p $prisoner
  ...
fi
```

Si el usuario ingresa datos antes de que expire el tiempo de espera, `read` devuelve 0 (éxito) y se procesa la parte `then` del `if`. Por otro lado, cuando el usuario no ingresa nada, el tiempo de espera expira y `read` devuelve 1 (fracaso), ejecutando la parte `else` de la declaración.

Aunque no es una opción para el comando *read*, la variable `TMOUT` puede afectarlo. Al igual que para `select`, si `TMOUT` está configurado con un número que representa cierta cantidad de segundos, el comando *read* se agota si no se ingresa nada dentro de ese tiempo, y devuelve un estado de salida de fracaso. La opción `-t` anula la configuración de `TMOUT`.

Finalmente, la opción `-un` es útil en scripts que leen de más de un archivo al mismo tiempo.

La Tarea 7-4 es un ejemplo de esto que también utiliza el redireccionador de entrada `n<` que vimos anteriormente en este capítulo.

Tarea 7-4

Escribe un script que imprima el contenido de dos archivos uno al lado del otro.

Formatearemos la salida para que las dos columnas tengan un ancho fijo de 30 caracteres. Aquí está el código:

```
typeset -L30 f1 f2
while read -u3 f1 && read -u4 f2; do
  print "$f1$f2"
done 3<$1 4<$2
```

`read -u3` lee del descriptor de archivo 3, y `3<$1` dirige el archivo dado como primer argumento para que sea la entrada en ese descriptor de archivo; lo mismo es válido para el segundo argumento y el descriptor de archivo 4. Recuerda que los descriptores de archivo 0, 1 y 2 ya se utilizan para la entrada/salida estándar. Usamos los descriptores de archivo 3 y 4 para nuestros dos archivos de entrada; es mejor comenzar desde 3 y trabajar hacia arriba hasta el límite del shell, que es 9.

El comando `typeset` y las comillas alrededor del argumento de `print` aseguran que las columnas de salida tengan 30 caracteres de ancho y que se preserve el espacio en blanco al final de las líneas del archivo. El bucle `while` lee una línea de cada archivo hasta que al menos uno de ellos se queda sin entrada.

Supongamos que el archivo *dave* contiene lo siguiente:

```
DAVE
Height: 177.8 cm.
Weight: 79.5 kg.
Hair: brown
Eyes: brown
```

Y el archivo *shirley* contiene esto:

```
SHIRLEY
Height: 167.6 cm.
Weight: 65.5 kg.
Hair: blonde
Eyes: blue
```

Si el script se llama *twocols*, entonces `twocols dave shirley` produce esta salida:

```
DAVE                SHIRLEY
Height: 177.8 cm.   Height: 167.6 cm.
Weight: 79.5 kg.    Weight: 65.5 kg.
Hair: brown         Hair: blonde
Eyes: brown         Eyes: blue
```

7.3. Procesamiento de la línea de comandos

Hemos visto cómo el shell procesa líneas de entrada: maneja comillas simples (' '), comillas dobles () y barras invertidas (\), y separa expansiones de parámetros, comandos y aritmética en palabras, según los delimitadores en la variable IFS. Esto es un subconjunto de las cosas que el shell hace al procesar líneas de comandos.

Esta sección completa la discusión, en ocasiones en detalle. Primero examinamos dos tipos adicionales de sustituciones o expansiones que el shell realiza y que pueden no estar disponibles de manera universal. Luego presentamos la historia completa del orden en que el shell procesa la línea de comandos. A continuación se aborda el uso de *comillas*, que evita que muchas o todas las etapas de sustitución ocurran. Finalmente, cubrimos el comando *eval*, que se puede utilizar para un control programático adicional de las evaluaciones de la línea de comandos.

7.3.1. Expansión de llaves y sustitución de procesos

La *expansión de llaves* es una característica tomada del intérprete de comandos Berkeley *csh* y también disponible en el popular shell *bash*. La expansión de llaves es una forma

de ahorrar escritura cuando tienes cadenas que son prefijos o sufijos entre sí. Por ejemplo, supongamos que tienes los siguientes archivos:

```
$ ls
cpp-args.c cpp-lex.c cpp-out.c cpp-parse.c
```

Podrías escribir `vi cpp-{args,lex,parse}.c` si deseas editar tres de los cuatro archivos C, y el shell expandiría esto a `vi cpp-args.c cpp-lex.c cpp-parse.c`. Además, las sustituciones de llaves pueden estar anidadas. Por ejemplo:

```
$ print cpp-{args,l{e,o}x,parse}.c
cpp-args.c cpp-lex.c cpp-lox.c cpp-parse.c
```

Esta es una característica útil. No la hemos cubierto hasta ahora porque es posible que tu versión de *ksh* no la tenga. Es una característica opcional que se habilita cuando se compila *ksh*. Sin embargo, se habilita de forma predeterminada cuando *ksh93* se compila desde el código fuente.

La sustitución de procesos te permite abrir múltiples flujos de procesos y alimentarlos en un único programa para su procesamiento. Por ejemplo:

```
awk '...' <(generate_data) <(generate_more_data)
```

(Ten en cuenta que los paréntesis son parte de la sintaxis; los escribes literalmente). Aquí, `generate_data` y `generate_more_data` representan comandos arbitrarios, incluidos conductos, que producen flujos de datos. El programa *awk* procesa cada flujo sucesivamente, sin darse cuenta de que los datos provienen de múltiples fuentes. Esto se muestra gráficamente en la Figura 7.1.a.

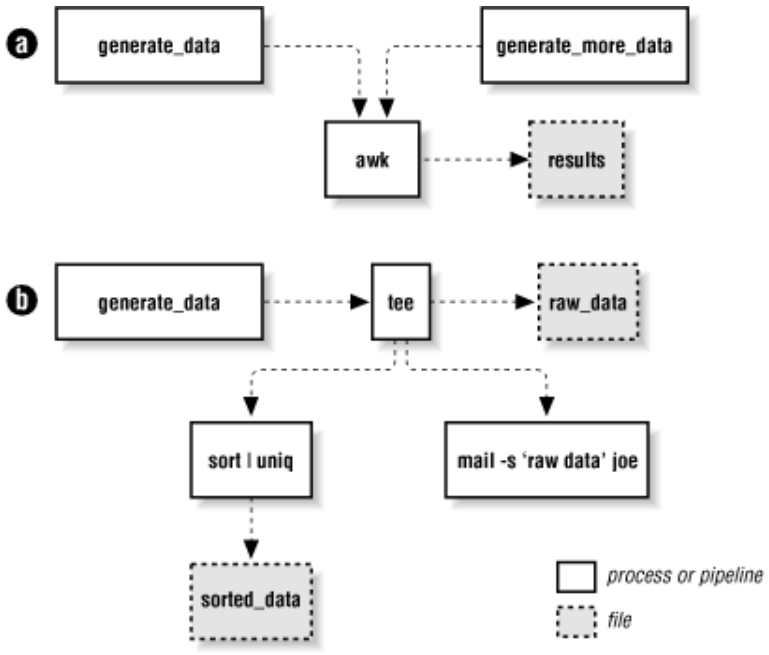

Figura 7.1: Sustitución de procesos para flujos de datos de entrada y salida

La sustitución de procesos también se puede utilizar para la salida, especialmente cuando se combina con el programa *tee(1)*, que envía su entrada a varios archivos de salida y a la salida estándar. Por ejemplo:

```
generate_data | tee >(sort | uniq > sorted_data) \
>(mail -s 'raw data' joe) > raw_data
```

Este comando utiliza *tee* para (1) enviar los datos a una tubería que ordena y guarda los datos, (2) enviar los datos al programa de correo (*mail*) al usuario *joe*, y (3) redirigir los datos originales a un archivo. Esto se representa gráficamente en la Figura 7.1.b. La sustitución de procesos, combinada con *tee*, te permite crear gráficos de datos no lineales, liberándote del paradigma lineal tradicional de «una entrada, una salida» de las tuberías Unix convencionales.

La sustitución de procesos solo está disponible en sistemas Unix que admiten archivos especiales `/dev/fd/N` para acceder de manera nombrada a descriptores de archivos ya abiertos. (Esto difiere del uso de `/dev/fd/N` descrito anteriormente en este capítulo, donde la propia shell interpreta la ruta de acceso. Aquí, debido a que los comandos externos deben poder abrir archivos en `/dev/fd`, la característica debe ser compatible directamente con el sistema operativo.) La mayoría de los sistemas Unix modernos, incluidos GNU/Linux, admiten esta característica. Al igual que la expansión de llaves, debe habilitarse en tiempo de compilación y es posible que no esté disponible en tu versión de *ksh*. Al igual que la expansión de llaves, se habilita de forma predeterminada cuando *ksh93* se compila a partir del código fuente.

7.3.2. Orden de sustitución

Hemos abordado el procesamiento de líneas de comandos (ver Figura 7.2) a lo largo de este libro; ahora es un buen momento para explicar todo el proceso. [12] Cada línea que el shell lee desde la entrada estándar o un script se denomina una *tubería*; contiene uno o más comandos separados por cero o más caracteres de tubería (|). Para cada tubería que lee, el shell la descompone en comandos, configura la E/S para la tubería y luego realiza lo siguiente para cada comando:

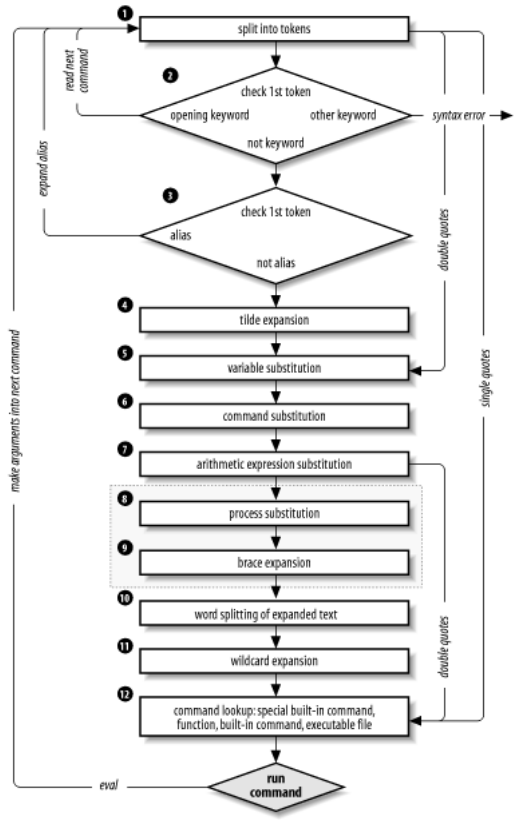

Figura 7.2: Pasos en el procesamiento de la línea de comandos

1. Divide el comando en tokens separados por el conjunto fijo de metacaracteres: espacio, TAB, nueva línea, ;, (,), <, >, |, y &. Los tipos de tokens incluyen palabras, palabras clave, redireccionadores de E/S y puntos y coma.

2. Verifica si el primer token de cada comando es una *palabra clave* sin comillas ni barras invertidas. Si es una palabra clave de apertura (como if y otros iniciadores de estructuras de control, `function`, `{`, `(`, `((`, o `[[`), el comando es en realidad un *comando compuesto*. El shell configura internamente las cosas para el comando

[12] Incluso esta explicación está ligeramente simplificada para omitir los detalles más pequeños, por ejemplo, «medios» y «finales» de comandos compuestos, caracteres especiales dentro de construcciones `[[...]]` y `((...))`, etc. La última palabra sobre este tema se encuentra en el libro de referencia *The New KornShell Command and Programming Language* de Morris Bolsky y David Korn, publicado por Prentice-Hall.

compuesto, lee el próximo comando y comienza el proceso nuevamente. Si la palabra clave no es una abertura de comando compuesto (por ejemplo, es una «parte» de una estructura de control como `then, else` o `do`, un «final» como `fi` o `done`, o un operador lógico), el shell señala un error de sintaxis.

3. Verifica la primera palabra de cada comando en la lista de *alias*. Si se encuentra una coincidencia, sustituye la definición del alias y vuelve al Paso 1; de lo contrario, pasa al Paso 4. Este esquema permite alias recursivos; consulta el Capítulo 3. También permite definir alias para palabras clave, por ejemplo, `alias aslongas=while` o `alias procedure=function`.

4. Sustituye el directorio principal del usuario ($HOME) por el carácter de tilde (~) si está al principio de una palabra. Sustituye el directorio principal del usuario por *~user*. [13].

   La sustitución de tilde ocurre en los siguientes lugares:

   - Como el primer carácter no entre comillas de una palabra en la línea de comandos.
   - Después del signo = en una asignación de variable y después de cualquier : en el valor de una asignación de variable.
   - Para la parte de *palabra* de las sustituciones de variables de la forma `${variable op palabra}` (ver Capítulo 4).

5. Realiza la sustitución de parámetros (variables) para cualquier expresión que comience con un signo de dólar ($)

6. Realiza la sustitución de comandos para cualquier expresión de la forma `$(cadena)` o `cadena`.

7. Evalúa expresiones aritméticas de la forma `$((cadena))`.

8. Realiza la sustitución de procesos, si esa característica está compilada en el shell y tu sistema admite `/dev/fd`.

9. Realiza la expansión de llaves, si esa característica está compilada en el shell.

10. Toma las partes de la línea que resultaron de la sustitución de parámetros, comandos y aritmética, y las divide nuevamente en palabras. Esta vez utiliza los caracteres en

[13] Dos variaciones oscuras de esto: el shell sustituye el directorio actual (`$PWD`) por `~+` y el directorio anterior (`$OLDPWD`) por `~-`

`$IFS` como delimitadores en lugar del conjunto de metacaracteres en el Paso 1.

Normalmente, las ocurrencias sucesivas múltiples de caracteres en IFS actúan como un único delimitador, como se esperaría. Esto es cierto solo para caracteres de espacio en blanco, como espacio y TAB. Para caracteres que no son de espacio en blanco, esto no es cierto. Por ejemplo, al leer los campos separados por dos puntos de `/etc/passwd`, dos dos puntos sucesivos delimitan un campo vacío. Por ejemplo:

```
IFS=:
while read name passwd uid gid fullname homedir shell
do
...
done < /etc/passwd
```

Para obtener este comportamiento con campos delimitados por espacios en blanco (por ejemplo, donde los caracteres de TAB delimitan cada campo), coloca dos instancias sucesivas del carácter delimitador en `IFS`.

ksh ignora cualquier valor heredado (del entorno) de `IFS`. Al iniciar, establece el valor de `IFS` en el valor predeterminado de espacio, TAB y nueva línea.

11. Realiza la generación de nombres de archivo, también conocida como expansión de comodines, para cualquier aparición de *, ?, y pares [ ]. También procesa los operadores de expresiones regulares que vimos en el Capítulo 4.

12. Utiliza la primera palabra como un comando buscando su ubicación según el resto de la lista en el Capítulo 4, es decir, como un comando especial incorporado, luego como una función, luego como un comando incorporado regular y, finalmente, como un archivo en cualquiera de los directorios en `$PATH`.

13. Ejecuta el comando después de configurar la redirección de E/S y otras cosas similares.

Eso son muchos pasos, ¡y ni siquiera es toda la historia! Pero antes de continuar, un ejemplo debería hacer este proceso más claro. Supongamos que se ha ejecutado el siguiente comando:

```
alias ll="ls -l"
```

Supongamos además que existe un archivo llamado *.hist537* en el directorio personal del usuario *fred*, que es `/home/fred`, y que hay una variable de doble signo de dólar $$ cuyo valor es 2537 (veremos qué es esta variable especial en el próximo capítulo).

Ahora veamos cómo el shell procesa el siguiente comando:

```
ll $(whence cc) ~fred/.*$(($$%1000))
```

Aquí está lo que sucede con esta línea:

1. `ll $(whence cc) ~fred/.*$(($$%1000))`
   División de la entrada en palabras.
2. `ll` no es una palabra clave, por lo que el paso 2 no hace nada.
3. `ls -l $(whence cc) ~fred/.*$(($$%1000))`
   Sustituyendo `ls -l` por su alias *ll*. Luego, el shell repite los pasos 1 al 3; el paso 2 divide `ls -l` en dos palabras.
4. `ls -l $(whence cc) /home/fred/.*$(($$%1000))`
   Expansión de `~fred` a `/home/fred`.
5. `ls -l $(whence cc) /home/fred/.*$((2537%1000))`
   Sustituyendo 2537 por $$.
6. `ls -l /usr/bin/cc /home/fred/.*$((2537%1000))`
   Haciendo la sustitución de comandos en `whence cc`.
7. `ls -l /usr/bin/cc /home/fred/.*537`
   Evaluando la expresión aritmética 2537 %1000.
8. `ls -l /usr/bin/cc /home/fred/.*537`
   Este paso no hace nada. (No hay sustitución de procesos.)
9. `ls -l /usr/bin/cc /home/fred/.*537`
   Este paso no hace nada. (No hay llaves para expandir.)
10. `ls -l /usr/bin/cc /home/fred/.*537`
    Este paso no hace nada. (No hay texto expandido para dividir.)
11. `ls -l /usr/bin/cc /home/fred/.hist537`
    Sustituyendo el nombre de archivo por la expresión comodín .*537.
12. El comando *ls* se encuentra en `/usr/bin`.
13. Se ejecuta `/usr/bin/ls` con la opción `-l` y los dos argumentos.

Aunque esta lista de pasos es bastante directa, no es toda la historia. Todavía hay dos formas de subvertir el proceso: mediante comillas y mediante el comando *eval* avanzado.

7.3.3. Citando

Puedes pensar en las comillas como una forma de hacer que el shell omita algunos de los 13 pasos mencionados anteriormente. En particular:

Las comillas simples (`'...'`) evitan *todo* hasta el Paso 11, incluyendo la expansión de alias. Todos los caracteres dentro de un par de comillas simples permanecen sin cambios. No puedes tener comillas simples dentro de comillas simples, incluso si las precedes con barras invertidas.[14] (es decir, comilla simple, barra invertida, comilla simple, comilla simple) actúa de manera bastante similar a una comilla simple en el medio de una cadena entre comillas simples; por ejemplo, `'abc'\''def'` se evalúa como `abc'def`

Las comillas dobles (`"..."`) evitan los pasos 1 al 4, más los pasos 8 al 11. Es decir, ignoran los caracteres de tubería, los alias, la sustitución de tilde, la expansión de comodines, la sustitución de procesos, la expansión de llaves y la división en palabras mediante delimitadores (por ejemplo, espacios) dentro de las comillas dobles. Las comillas simples dentro de las comillas dobles no tienen ningún efecto. Pero las comillas dobles permiten la sustitución de parámetros, la sustitución de comandos y la evaluación de expresiones aritméticas. Puedes incluir una comilla doble dentro de una cadena entre comillas dobles precediéndola con una barra invertida (\). También debes escapar con barra invertida los caracteres `$, ` ` (el delimitador arcaico de sustitución de comandos) y la barra invertida en sí.

La Tabla 7.8 contiene algunos ejemplos simples que muestran cómo funcionan estas reglas; asumen que se ejecutó la declaración `dave=bob` y que el directorio personal del usuario *fred* es `/home/fred`.

Si te preguntas si debes usar comillas simples o dobles en una situación específica de programación de shell, es más seguro usar comillas simples a menos que necesites específicamente la sustitución de parámetros, comandos o aritmética.

Tabla 7.8: Ejemplos de reglas de comillas

Expresión	Valor
`$dave`	`bob`
`"$dave"`	`bob`

[14] Sin embargo, como vimos en el Capítulo 1, `['\"`

Expresión	Valor
`\$dave`	`$dave`
`'$dave'`	`$dave`
`\'$dave\'`	`'bob'`
`~fred`	`/home/fred`
`"~fred"`	`~fred`
`'~fred'`	`~fred`

El uso de comillas dobles en los valores de las variables es cada vez más importante al tratar con los resultados de la expansión de comodines. Hoy en día, no es inusual tener archivos y directorios disponibles en sistemas Unix que físicamente existen en sistemas Microsoft Windows y Apple Macintosh. En esos sistemas, los espacios y otros caracteres inusuales, como apóstrofes y comillas invertidas, son comunes en los nombres de archivo. Por lo tanto, para pasar la ruta completa a tu aplicación, asegúrate de citar las cosas adecuadamente.

La Tarea 7-5 es un ejemplo más avanzado de procesamiento de línea de comandos que debería brindarte una comprensión más profunda del proceso general.

Tarea 7-5

Personaliza tu cadena de prompt principal para que contenga el directorio actual con notación de tilde (~).

Recuerda del Capítulo 4 que encontramos una forma simple de configurar la cadena de prompt `PS1` para que siempre contenga el directorio actual: `PS1='($PWD)-> '`.

Un problema con esta configuración es que las cadenas de prompt resultantes pueden volverse muy largas. Una forma de acortarlas es sustituir la notación de tilde por los directorios principales de los usuarios. Esto no se puede hacer con una simple expresión de cadena análoga a la de arriba. La solución es algo complicada y aprovecha las reglas de procesamiento de la línea de comandos.

La idea básica es crear una «envoltura» alrededor del comando *cd*, como hicimos en el Capítulo 5, que instale el directorio actual con notación de tilde como la cadena de prompt. Veremos cómo crear esta función de envoltura en breve. El código que necesitamos para insertar la notación de tilde es complicado por sí mismo; lo desarrollamos primero.

Comenzamos con una función que, dada una ruta como argumento, imprime su equivalente

en notación de tilde si es posible. Para escribir esta función, asumimos que ya tenemos una matriz asociativa llamada `tilde_ids`, en la que los subíndices son directorios principales y los valores son nombres de usuario. Por lo tanto, `print ${tilde_ids[/home/arnold]}` imprimiría el valor `arnold`. Aquí está la función, llamada *tildize*:

```
function tildize {
  # subdirectorio de nuestro directorio principal
  if [[ $1 == $HOME* ]]; then
    print "\~${1#$HOME}"
    return 0
  fi

  # bucle sobre los directorios principales intentando hacer coincidir el directorio actual
  typeset homedir
  for homedir in ${!tilde_ids[*]}; do
    if [[ $1 == ${homedir}?(/*) ]]; then
      print "\~${tilde_ids[$homedir]}${1#$homedir}"
      return 0
    fi
  done
  print "$1"
  return 1
}
```

La primera cláusula `if` verifica si la ruta dada está bajo el directorio principal del usuario. Si es así, sustituye la tilde (~) por el directorio principal en la ruta y devuelve.

Si no es así, hacemos un bucle sobre todos los subíndices en `tilde_ids`, comparando cada uno con nuestro directorio actual. La prueba hace coincidir los directorios principales por sí mismos o con algún otro directorio añadido (la parte `?(/*)`). Si se encuentra el directorio principal de un usuario, `~usuario` se sustituye por el directorio principal completo en la ruta dada, se imprime el resultado y la función sale.

Finalmente, si el bucle `for` agota todos los usuarios sin encontrar un directorio principal que sea un prefijo de la ruta dada, `tildize` simplemente devuelve su entrada.

Ahora, ¿cómo creamos la matriz `tilde_ids`? Usamos la función `init_tilde_db`. Debe llamarse una vez, desde el archivo *.profile* cuando iniciamos sesión. La matriz `tilde_ids` debe declararse explícitamente como una matriz asociativa utilizando typeset `-A`:

```
# tilde_ids[] es una matriz asociativa global
# que asigna directorios a nombres de usuario
typeset -A tilde_ids

function init_tilde_db {
  typeset user homedir      # variables locales
  awk -F: '{ print $1, $6 }' /etc/passwd |
```

```
    while read user homedir; do
       if [[ $homedir != / ]]; then
          tilde_ids[$homedir]=$user
       fi
    done
}
```

Usamos la utilidad *awk* para extraer los primeros y sextos campos del archivo `/etc/passwd`, que contienen IDs de usuario y directorios principales, respectivamente.[15] En este caso, *awk* actúa como *cut*. El `-F:` es análogo a `-d:`, que vimos en el Capítulo 4, excepto que *awk* imprime los valores en cada línea separados por espacios, no por dos puntos (:).

El resultado de *awk* se alimenta a un bucle `while` que verifica la ruta dada como argumento para ver si contiene el directorio principal de algún usuario. (La expresión condicional elimina «usuarios» como `daemon` y `root`, cuyos directorios principales son root y, por lo tanto, están contenidos en cada ruta completa).

Ahora que tenemos la función `tildize`, podrías pensar que podríamos usarla en una expresión de sustitución de comandos de la siguiente manera:

```
PS1='$(tildize $PWD)> '
```

De hecho, estarías en lo correcto.[16] Pero hay un costo oculto aquí. La función se ejecuta *cada vez* que el shell imprime el prompt. Incluso si solo presionas ENTER, el shell ejecuta la función *tildize*. Si hay muchos usuarios en tu sistema, el shell recorre todos los directorios principales cada vez. Para evitar esto, escribimos una función *cd* que solo actualiza el prompt cuando realmente cambiamos de directorio. El siguiente código debería ir en tu archivo *.profile* o de entorno, junto con la definición de `tilde_ids` y `tildize`:

```
init_tilde_db # configurar la matriz una vez, al iniciar sesión

function cd {
  command cd "$@"     # ejecutar el comando cd real
  typeset es=$?     # guardar el estado de salida en una variable local
  PS1="$(tildize $PWD)> "
  return $es
}

cd $PWD # establecer el prompt
```

Como vimos en el Capítulo 5, escribir una función con el mismo nombre que un comando incorporado parece bastante extraño a primera vista. Pero, siguiendo el estándar POSIX, el

[15] En entornos grandes con varias máquinas, puede que necesites usar algo como `ypcat passwd | awk ...` o `niscat passwd.org_dir | awk ...` para obtener la misma información. Consulte con el administrador de su sistema.

[16] Sin embargo, esto no funciona en *ksh88*.

shell de Korn distingue entre comandos incorporados «especiales» y comandos incorporados regulares. Cuando el shell busca comandos para ejecutar, encuentra funciones antes de encontrar comandos incorporados regulares. *cd* es un comando incorporado regular, así que esto funciona. Dentro de la función, usamos el comando ingeniosamente llamado *command* para acceder realmente al comando *cd* real.[17] La declaración `command cd "$@"` pasa los argumentos de la función al *cd* real para cambiar el directorio. (Como nota al margen, el shell define un alias `command='command '`, lo que te permite usar `command` con alias).

Cuando inicias sesión, este código establece `PS1` en el directorio actual inicial (presumiblemente tu directorio principal). Luego, cada vez que ingresas un comando *cd*, la función se ejecuta para cambiar el directorio y restablecer el prompt.

Por supuesto, la función `tildize` puede ser cualquier código que formatee la cadena del directorio. Consulta los ejercicios al final de este capítulo para obtener algunas sugerencias.

Cita ampliada

Las comillas simples y dobles han estado presentes en el shell de Bourne y sus derivados desde el principio (aunque el shell de Bourne original no realiza aritmética ni sustitución de `$(...)`). El shell de Korn ofrece versiones variantes de cadenas tanto con comillas simples como dobles, de la siguiente manera.

`$"..."` Esta versión es la más simple. Es similar a una cadena de comillas dobles regular. Sin embargo, estas cadenas están sujetas a la *traducción de localización* en tiempo de ejecución. Esto se describe más adelante, a continuación.

`$'...'` Esta cadena es similar a una cadena de comillas simples regular en el sentido de que no se realizan sustituciones ni expansiones del shell en el contenido. Sin embargo, el contenido se procesa en busca de secuencias de escape, similares a las utilizadas por el comando *print*. La documentación de *ksh* se refiere a estas como cadenas ANSI C.

Las características de internacionalización del shell del Korn están más allá del alcance de este libro, pero brevemente, funciona así. Cuando se invoca *ksh* en un script con la opción `-D`, imprime una lista de todas las cadenas `$"..."` en la salida estándar. Esta lista puede guardarse y usarse para producir traducciones que se utilizan en tiempo de ejecución cuando el script se ejecuta realmente. Así, en una configuración regional francesa, si hay

[17] Como se mencionó anteriormente, `command` no es un comando incorporado especial. ¡Ay del programador del shell que escriba una función llamada `command!`

una traducción disponible para este programa:

```
print $"hello, world" Un saludo bien conocido entre los científicos de la computación
```

ksh imprimiría bonjour, monde cuando se ejecuta el programa.

El comando *print* permite utilizar secuencias de escape de estilo C para la salida. Y la mayoría de las veces, esto es todo lo que necesitas. Pero ocasionalmente, es útil utilizar la misma notación en los argumentos de otros programas. Este es el propósito de la cadena `$'...'`. El contenido no se procesa para la sustitución de variables, comandos o aritmética. Pero sí se procesan las secuencias de escape, como se muestra en la Tabla 7.9.

Tabla 7.9: Secuencias de escape de cadenas

Secuencia	**Significado**	**Secuencia**	**Significado**
\a	Alerta, Campana ASCII	\t	TAB
\b	Retroceso	\v	Tabulación vertical
\x*X*	CTRL-*X* [1] [2]	\x*HH*	Carácter con valor de dígitos hexadecimales *HH*
\C[.*ce*.]	El elemento de intercambio *ce* [1] [2] (Un elemento de intercalación son dos o más carcteres que se tratan como una unidad a efectos de clasificación).	\x{*digs*}	Valor hexadecimal de los dígitos. Utilice los corchetes cuando los siguientes caracteres sean dígitos hexadecimales que no deban interpretarse. [1] [2]
\e	Carácter de escape ASCII [1] [2]	\0	El resto de la cadena se ignora [2]
\E	Carácter de escape ASCII [1]	*ddd*	Carácter con valor de dígitos octales *ddd*
\f	Forma de alimentación	\’	Comillas simples
\n	Nueva línea	\”	Comillas doble
\r	Retorno de carro	\\	Barra invertida literal

De valor primordial es el hecho de que puede obtener fácilmente comillas simples y dobles dentro del tipo de cadena `$'...'`:

```
$ print $'Una cadena con \'comillas simples\' y \"comillas dobles\" en ella'
Una cadena con 'comillas simples' y "comillas dobles" en ella
```

Es interesante el hecho de que las comillas dobles no necesitan ser escapadas, pero que hacerlo tampoco hace daño.

[1] No en el lenguaje C.
[2] Nuevo, empezando por *ksh93l*.

7.3.4. eval

Hemos visto que las comillas te permiten omitir pasos en el procesamiento de la línea de comandos. Luego está el comando *eval*, que te permite repetir el proceso. Realizar el procesamiento de la línea de comandos dos veces puede parecer extraño, pero en realidad es muy potente: te permite escribir scripts que crean cadenas de comandos sobre la marcha y luego las pasan al shell para su ejecución. Esto significa que puedes dar a los scripts «inteligencia» para modificar su propio comportamiento mientras se están ejecutando.

La instrucción *eval* le indica al shell que tome los argumentos de *eval* y los vuelva a ejecutar a través de los pasos de procesamiento de la línea de comandos. Para ayudarte a entender las implicaciones de *eval*, comenzaremos con un ejemplo trivial y avanzaremos hacia una situación en la que estamos construyendo y ejecutando comandos sobre la marcha.

`eval ls` pasa la cadena `ls` al shell para que la ejecute; el shell imprime una lista de archivos en el directorio actual. Muy simple; no hay nada en la cadena `ls` que necesite pasar por los pasos de procesamiento de comandos dos veces. Pero considera esto:

```
listpage="ls | more"
$listpage
```

En lugar de producir una lista de archivos paginada, el shell trata | y `more` como argumentos de *ls*, y *ls* se queja de que no existen archivos con esos nombres. ¿Por qué? Porque el carácter de tubería «aparece» en el paso 5 cuando el shell evalúa la variable, *después* de que realmente ha buscado los caracteres de tubería (en el paso 2). La expansión de la variable ni siquiera se analiza hasta el paso 10. Como resultado, el shell trata | y *more* como argumentos de *ls*, de modo que *ls* intenta encontrar archivos llamados | y *more* en el directorio actual.

Ahora considera `eval $listpage` en lugar de simplemente `$listpage`. Cuando el shell llega al último paso, ejecuta el comando *eval* con los argumentos *ls*, | y *more*. Esto hace que el shell vuelva al Paso 1 con una línea que consiste en estos argumentos. Encuentra | en el Paso 2 y divide la línea en dos comandos, *ls* y *more*. Cada comando se procesa de la manera normal (y en ambos casos de manera trivial). El resultado es una lista paginada de los archivos en tu directorio actual.

Ahora es posible que empieces a ver cuán poderoso puede ser *eval*. Es una característica avanzada que requiere una considerable astucia de programación para usarse de manera más efectiva. Incluso tiene un poco del sabor de la inteligencia artificial, ya que te permite

escribir programas que pueden «escribir» y ejecutar otros programas. [18] Es probable que no uses *eval* para la programación diaria en el shell, pero vale la pena tomarse el tiempo para entender lo que puede hacer.

Como ejemplo más interesante, revisaremos la Tarea 4-1, la primera tarea del libro. En ella, construimos una canalización simple que ordena un archivo e imprime las primeras *N* líneas, donde N es 10 por defecto. La canalización resultante fue:

```
sort -nr $1 | head -${2:-10}
```

El primer argumento especifica el archivo a ordenar; `$2` es el número de líneas a imprimir.

Ahora supongamos que cambiamos un poco la tarea para que, en lugar de 10 líneas, la impresión predeterminada sea de *todo el archivo*. Esto significa que no queremos usar *head* en absoluto en el caso predeterminado. Podríamos hacer esto de la siguiente manera:

```
if [[ -n $2 ]]; then
    sort -nr $1 | head -$2
else
    sort -nr $1
fi
```

En otras palabras, decidimos qué canalización ejecutar según si `$2` es nulo o no. Pero aquí hay una solución más compacta:

```
eval sort -nr \$1 ${2:+"| head -\$2"}
```

La última expresión en esta línea evalúa a la cadena `| head -\$2` si `$2` existe (no es nulo); si `$2` es nulo, entonces la expresión también es nula. Escapamos los signos de dólar (`\$`) antes de los nombres de las variables para evitar resultados impredecibles si los valores de las variables contienen caracteres especiales como > o |. El backslash efectivamente pospone la evaluación de las variables hasta que se ejecuta el comando *eval* en sí. Entonces, toda la línea es ya sea:

```
eval sort -nr \$1 | head -\$2
```

si `$2` está dado o:

```
eval sort -nr \$1
```

si `$2` es nulo. Una vez más, no podemos simplemente ejecutar este comando sin *eval* porque el tubo está «descubierto» después de que el shell intenta dividir la línea en comandos. *eval* hace que el shell ejecute la canalización correcta cuando se proporciona `$2`.

[18] De hecho, podrías hacer esto sin *eval*, imprimiendo comandos en un archivo temporal y luego «sourcing» ese archivo con `. filename`. Pero eso es mucho menos eficiente.

A continuación, revisaremos la Tarea 7-3 de antes en este capítulo, la función *start* que te permite iniciar un comando en segundo plano y guardar su salida estándar y su error estándar en un archivo de registro. Recuerda que la solución de una sola línea para esta tarea tenía la restricción de que el comando no podía contener redireccionadores de salida o tuberías. Aunque lo primero no tiene sentido cuando lo piensas, ciertamente querrías la capacidad de iniciar una tubería de esta manera.

eval es la forma obvia de resolver este problema:

```
function start {
    eval "$@" > logfile 2>&1 &
}
```

La única restricción que esto impone al usuario es que las tuberías y otros caracteres especiales deben estar entre comillas o precedidos por barras invertidas.

La Tarea 7-6 es una manera de aplicar *eval* junto con varios otros conceptos interesantes de programación de shell.

Tarea 7-6

Implementa la esencia del programa *make(1)* como un script de shell.

make se conoce principalmente como una herramienta para programadores, pero parece que alguien encuentra un nuevo uso para ella todos los días. Sin entrar en demasiados detalles innecesarios, *make* lleva un seguimiento de múltiples archivos en un proyecto particular, algunos de los cuales dependen de otros (por ejemplo, un documento depende de su(s) archivo(s) de entrada del procesador de texto). Se asegura de que cuando cambias un archivo, todos los demás archivos que dependen de él se procesen.

Por ejemplo, supongamos que estás escribiendo un libro en DocBook XML. Tienes archivos para los capítulos del libro llamados *ch01.xml, ch02.xml*, y así sucesivamente. La salida generada en PostScript para estos archivos es *ch01.ps, ch02.ps*, etc. La herramienta para convertir DocBook XML en PostScript se llama (por alguna razón extraña) *gmat*. Ejecutas comandos como `gmat chN.xml` para realizar el procesamiento. (*gmat* sabe crear *ch01.ps* a partir de *ch01.xml*; no necesitas usar la redirección de shell). Mientras trabajas en el libro, tiendes a realizar cambios en varios archivos al mismo tiempo.

En esta situación, puedes usar *make* para llevar un seguimiento de qué archivos deben ser reprocesados, de modo que todo lo que necesitas hacer es escribir `make`, y determina qué debe hacerse. No necesitas recordar reprocesar los archivos que han cambiado.

¿Cómo lo hace *make*? Simple: compara los tiempos de modificación de los archivos de entrada y salida (llamados *fuentes* y *objetivos* en la terminología de *make*), y si el archivo de entrada es más reciente, *make* lo vuelve a procesar.

Le dices a *make* qué archivos verificar construyendo un archivo llamado *makefile* que tiene construcciones como esta:

```
target : source1 source2 ...
    commands to make target
```

Esto esencialmente dice: «Para que el objetivo esté actualizado, debe ser más nuevo que todas las *fuentes*. Si no lo es, ejecuta los *comandos* para ponerlo al día». Los *comandos* están en una o más líneas que deben comenzar con TABs; por ejemplo, para hacer *ch07.ps*:

```
ch07.ps : ch07.xml
    gmat ch07.xml
```

Ahora supongamos que escribimos una función de shell llamada *makecmd* que lee y ejecuta una sola construcción de esta forma. Supongamos que el *makefile* se lee desde la entrada estándar. La función se vería así en el siguiente código.

```
function makecmd {
    read target colon sources
    for src in $sources; do
        if [[ $src -nt $target ]]; then
            while read cmd && [[ $cmd == \t* ]]; do
                print "$cmd"
                eval $cmd
            done
            break
        fi
    done
}
```

Esta función lee la línea con el objetivo y las fuentes; la variable `colon` es solo un marcador de posición para el `:`. Luego verifica cada fuente para ver si es más nueva que el objetivo, utilizando el operador de prueba de atributo de archivo `-nt` que vimos en el Capítulo 5. Si la fuente es más nueva, lee, imprime y ejecuta los comandos hasta que encuentra una línea que no comienza con un TAB o llega al final del archivo. (El *make* real hace más que esto; consulta los ejercicios al final de este capítulo). Después de ejecutar los comandos, sale del bucle `for`, para que no ejecute los comandos más de una vez. (No es necesario quitar el TAB inicial del comando. El shell descarta automáticamente los espacios en blanco iniciales).

El compilador C como tuberia (pipeline)

Como ejemplo final de *eval*, revisaremos nuestro viejo amigo *occ*, el front-end del compilador C de los tres capítulos anteriores. Recuerda que el front-end del compilador realiza su trabajo llamando a programas separados para realizar la compilación real de C a código objeto (el programa *ccom*), la optimización del código objeto (`optimize`), el ensamblaje de archivos de código ensamblador (*as*), y la vinculación final de archivos de código objeto en un programa ejecutable (*ld*). Estos programas separados utilizan archivos temporales para almacenar sus salidas.

Ahora asumiremos que estos componentes (excepto el enlazador) pasan información en una tubería (pipeline) a la salida final de código objeto. En otras palabras, cada componente toma la entrada estándar y produce la salida estándar en lugar de tomar argumentos de nombre de archivo. También cambiaremos una suposición anterior: en lugar de compilar directamente un archivo fuente C a código objeto, *occ* compila C a código ensamblador, que el ensamblador luego ensambla a código objeto.[19] Esto nos permite suponer que *occ* funciona así:

```
ccom < filename.c | as | optimize > filename.o
```

O, si prefieres:

```
cat filename.c | ccom | as | optimize > filename.o
```

Para ajustar esto al marco adecuado para *eval*, asumamos que las variables `srcname` y `objname` contienen los nombres de los archivos fuente y objeto, respectivamente. Entonces, nuestra tubería se convierte en:

```
cat $srcname | ccom | as | optimize > $objname
```

Como ya hemos visto, esto es equivalente a:

```
eval cat \$srcname \| ccom \| as \| optimize \> \$objname
```

Sabiendo lo que sabemos sobre *eval*, podemos transformar esto en:

```
eval cat \$srcname " | ccom" " | as" " | optimize" \> \$objname
```

y a partir de eso en:

```
compile=" | ccom"
assemble=" | as"
optimize=" | optimize"
```

[19] Por si sirve de algo, muchos compiladores Unix generan código ensamblador, optimizan el código ensamblador y luego generan código objeto.

```
eval cat \$srcname $compile $assemble $optimize \> \$objname
```

Ahora, considera qué sucede si no quieres invocar al optimizador, que es el caso predeterminado de todos modos. (Recuerda que la opción `-O` invoca al optimizador.) Podemos hacer esto:

```
optimize=""
if -O given then
  optimize=" | optimize"
fi
```

En el caso predeterminado, `$optimize` evalúa a una cadena vacía, haciendo que la tubería final *colapse* a:

```
eval cat $srcname \| ccom \| as \> $objname
```

De manera similar, si pasas a *occ* un archivo de código ensamblador (*filename.s*), puedes simplificar el paso de compilación:[20]

```
assemble="| as"
if [[ $srcname ends in .s ]]; then
    compile=""
fi
```

Eso resulta en esta tubería:

```
eval cat \$srcname \| as \> \$objname
```

Ahora estamos listos para mostrar la versión completa de «pipeline» de *occ*. Es similar a la versión anterior, excepto que para cada archivo de entrada, construye y ejecuta una tubería como se describió anteriormente. Procesa la opción `-g` (depuración) y el paso de enlace de la misma manera que antes. Aquí está el código:

```
# initialize option-related variables
do_link=true
debug=""
link_libs=""
clib="-lc"
exefile=""

# initialize pipeline components
compile=" | ccom"
assemble=" | as"
optimize=""
```

[20]Los lectores astutos se darán cuenta de que, según este razonamiento, manejaríamos los archivos de entrada de código objeto (*filename.o*) con la línea `eval cat $srcname > $objname`, donde los dos nombres son iguales. Esto hará que el shell destruya *filename.o* truncándolo a longitud cero. No nos preocuparemos por esto aquí.

```
# process command-line options
while getopts "cgl:[lib]o:[outfile]O files ..." opt; do
  case $opt in
    c ) do_link=false ;;
    g ) debug="-g" ;;
    l ) link_libs+=" -l $OPTARG" ;;
    o ) exefile="-o $OPTARG" ;;
    O ) optimize=" | optimize" ;;
  esac
done
shift $(($OPTIND - 1))

# process the input files
for filename in "$@"; do
  case $filename in
    *.c )
      objname=${filename%.c}.o ;;
    *.s )
      objname=${filename%.s}.o
      compile="" ;;
    *.o )
      objname=$filename # just link it directly with the rest
      compile=""
      assemble="" ;;
    * )
      print "error: $filename is not a source or object file."
      exit 1 ;;
  esac

  # run a pipeline for each input file
  eval cat \$filename $compile $assemble $optimize \> \$objname
  objfiles+=" $objname"
  compile=" | ccom"
  assemble=" | as"
done

if [[ $do_link == true ]]; then
  ld $exefile $objfiles $link_libs $clib
fi
```

Podríamos seguir indefinidamente con ejemplos cada vez más complejos de *eval*, pero nos conformaremos con concluir el capítulo con algunos ejercicios.

1. Aquí hay algunas formas de mejorar *occ*, nuestro compilador de C:

   - Los compiladores de C del mundo real aceptan la opción `-S`, que le indica al compilador que suprima el paso de ensamblaje y deje la salida en archivos de código ensamblador cuyos nombres terminan en `.s`. Modifica *occ* para que

reconozca esta opción.

- El lenguaje C++ es un sucesor evolutivo de C; incluye características avanzadas como la sobrecarga de operadores, la verificación obligatoria del tipo de argumento de función, definiciones de clases, plantillas y muchas más. (No te preocupes si no sabes lo que son). Algunos compiladores de C++ utilizan C como un «lenguaje ensamblador», es decir, compilan archivos fuente de C++ a código C y luego los pasan a un compilador de C para su procesamiento adicional. Supón que los archivos fuente de C++ tienen nombres que terminan en .cc y que `/lib/cfront` es el «front-end» del compilador C++ que produce código C en su salida estándar. Modifica *occ* para que acepte tanto C++ como C, archivos de código ensamblador y de objeto.

2. Las posibilidades de personalización de tu cadena de comandos del sistema son prácticamente infinitas. Aquí hay dos mejoras a los esquemas de personalización que hemos visto:

   - Mejora el esquema del directorio actual en la cadena de comandos del sistema limitando la longitud de la cadena de comandos a un número de caracteres que el usuario pueda definir con una variable de entorno.

   - Lee la página del manual de *date(1)* y lee acerca de la variable SECONDS en la página del manual de *ksh(1)*. Organiza las cosas de modo que el shell imprima la hora actual en la cadena de comandos del sistema. (Pista: recuerda que el shell realiza sustitución de variables, comandos y expresiones aritméticas en el valor de PS1 antes de imprimirlo).

3. La función *makecmd* en la solución de la Tarea 7-6 representa una simplificación de la funcionalidad real de *make*. *make* verifica las dependencias de archivos de manera *recursiva*, lo que significa que una fuente en una línea en un archivo *makefile* puede ser un objetivo en otra línea. Por ejemplo, los capítulos del libro en el ejemplo podrían depender de figuras en archivos separados que se hicieron con un paquete de gráficos.

   - Escribe una función llamada *readtargets* que recorra el archivo *makefile* y almacene todos los objetivos en una variable o archivo temporal.

   - En lugar de leer el archivo *makefile* desde la entrada estándar, léelo en una variable de matriz llamada `lines`. Usa la variable `curline` como el índice de «línea actual». Modifica *makecmd* para que lea líneas desde la matriz comenzando con

la línea actual.

- *makecmd* simplemente verifica si alguna de las fuentes es más reciente que el objetivo dado. Realmente debería ser una rutina recursiva que se ve así:

```
function makecmd {
  target=$1
  get sources for $target
  for each source src; do
    if $src is also a target in this makefile then
      makecmd $src
    fi
    if [[ $src -nt $target ]]; then
      run commands to make target
      return
    fi
  done
}
```

  Implementa esto. Recuerda usar *typeset* para crear variables locales, y piensa en cómo los arrays asociativos podrían ser útiles para hacer un seguimiento de los objetivos, las fuentes y los comandos a ejecutar.

- Escribe el script *driver* que convierta la función *makecmd* en un programa *make* completo. Esto debería hacer el objetivo dado como argumento, o si no se proporciona ninguno, el primer objetivo listado en el archivo *makefile*.

4. Finalmente, aquí hay algunos problemas que realmente ponen a prueba tu conocimiento de *eval* y las reglas de procesamiento de comandos del shell. ¡Resuelve estos y serás un verdadero mago de Korn shell!

   - Los programadores avanzados de shell a veces usan un pequeño truco que incluye *eval*: usar el valor de una variable como el nombre de otra variable. En otras palabras, puedes darle a un script de shell control sobre los nombres de las variables a las que asigna valores. ¿Cómo harías esto? (Pista: si `$fred` es igual a `'dave'`, y `$dave` es igual a `'bob'`, podrías pensar que podrías escribir `print $$fred` y obtener la respuesta *bob*. Esto no funciona realmente, pero va en la dirección correcta. Este ejercicio es fácil de resolver usando *namerefs*, pero vale la pena hacerlo sin ellas para probar tu comprensión de *eval* y las reglas de comillas del shell.)

   - Podrías usar la técnica anterior junto con otros trucos de *eval* para implementar nuevas estructuras de control para el shell. Por ejemplo, intenta escribir un script

(o función) que emule el comportamiento del comando *repeat* del C shell:

```
`repeat count command`
```

Esto funciona de manera obvia: el *comando* se ejecuta *count* veces.

CAPÍTULO 8

MANEJO DE PROCESOS

El sistema operativo Unix construyó su reputación sobre un pequeño número de conceptos, todos los cuales son simples pero poderosos. Ya hemos visto la mayoría de ellos: entrada/salida estándar, tuberías, utilidades de filtrado de texto, el sistema de archivos estructurado en árbol, y así sucesivamente. Unix también ganó notoriedad como el primer sistema operativo para computadoras pequeñas [1] que le daba a cada usuario control sobre más de un proceso. Llamamos a esta *capacidad multitarea controlada* por el usuario.

Si Unix es el único sistema operativo que te resulta familiar, podrías sorprenderte al descubrir que varios otros sistemas operativos importantes han carecido lamentablemente en esta área. Por ejemplo, el MS-DOS de Microsoft, para compatibles IBM PC, no tiene multitarea en absoluto, y mucho menos multitarea controlada por el usuario. El sistema VM/CMS de IBM para mainframes grandes maneja múltiples usuarios pero les otorga solo un proceso cada uno. OpenVMS de Compaq tiene multitarea controlada por el usuario, pero es limitada y difícil de usar. La última generación de sistemas operativos para computadoras pequeñas, como el Macintosh OS X de Apple (que se basa en BSD) y el Windows de Microsoft (Windows 95 y posteriores), finalmente incluyen la multitarea controlada por el usuario a nivel del sistema operativo.

Pero si has llegado tan lejos en este libro, probablemente no pienses que la multitarea sea algo importante. Probablemente estés acostumbrado a la idea de ejecutar un proceso en segundo plano colocando un ampersand (&) al final de la línea de comandos. También has visto la idea de un subproceso de shell en el Capítulo 4, cuando mostramos cómo se ejecutan los scripts de shell.

En este capítulo, cubrimos la mayoría de las características del shell Korn que se relacionan

[1] Los sistemas PDP-11 en los que Unix se volvió popular inicialmente se consideraban pequeños para esa época.

con la multitarea y el manejo de procesos en general. Decimos «la mayoría» porque algunas de estas características son, al igual que los descriptores de archivos que vimos en el Capítulo 7, de interés solo para programadores de sistemas de bajo nivel.

Comenzamos mirando ciertos primitivos importantes para identificar procesos y controlarlos durante las sesiones de inicio y dentro de los scripts de shell. Luego nos movemos a una perspectiva de nivel superior, examinando formas de hacer que los procesos se comuniquen entre sí. La facilidad de las corrutinas del shell Korn es el esquema de comunicación interprocesos más sofisticado que examinaremos; también analizamos en más detalle conceptos que ya hemos visto, como las tuberías y los subprocesos de shell.

No te preocupes por atascarte en detalles técnicos de bajo nivel sobre Unix. Proporcionamos solo la información técnica necesaria para explicar las características de nivel superior, además de algunos otros detalles diseñados para despertar tu curiosidad. Si estás interesado en obtener más información sobre estas áreas, consulta tu Manual del programador de Unix o un libro sobre internos de Unix que sea relevante para tu versión de Unix.

Recomendamos encarecidamente que pruebes los ejemplos en este capítulo. El comportamiento del código que involucra múltiples procesos no es tan fácil de entender en papel como la mayoría de los otros ejemplos en este libro.

8.1. ID de proceso y números de trabajo

Unix asigna números, llamados *identificadores de proceso* (PID, por sus siglas en inglés), a todos los procesos cuando se crean. Notarás que, cuando ejecutas un comando en segundo plano agregándole un ampersand (&), el shell responde con una línea que se ve así:

```
$ fred &
[1]   2349
```

En este ejemplo, 2349 es el PID del proceso *fred*. El `[1]` es un *número de trabajo* asignado por el shell (no el sistema operativo). ¿Cuál es la diferencia? Los números de trabajo se refieren a procesos en segundo plano que se están ejecutando actualmente bajo tu shell, mientras que los PID se refieren a todos los procesos que se están ejecutando actualmente en todo el sistema, para todos los usuarios. El término *job* básicamente se refiere a una línea de comando que se invocó desde tu shell de inicio de sesión.

Si inicias trabajos adicionales en segundo plano mientras el primero aún se está ejecutando, el shell los numera como 2, 3, etc. Por ejemplo:

```
$ bob &
[2]     2367
$ dave | george &
[3]     2382
```

Claramente, 1, 2 y 3 son más fáciles de recordar que 2349, 2367 y 2382.

El shell incluye números de trabajo en mensajes que imprime cuando se completa un trabajo en segundo plano, como este:

```
[1] + Done          fred &
```

Explicaremos lo que significa el signo más pronto. Si el trabajo sale con un estado distinto de cero (ver Capítulo 5), el shell incluye el estado de salida entre paréntesis:

```
[1] + Done(1)       fred &
```

El shell imprime otros tipos de mensajes cuando suceden ciertas cosas anormales con los trabajos en segundo plano; veremos estos más adelante en este capítulo.

8.2. Control de trabajo

¿Por qué deberías preocuparte por los identificadores de proceso o los números de trabajo? En realidad, podrías manejarte bien en tu vida de Unix sin referirte nunca a los identificadores de proceso (a menos que uses una estación de trabajo con ventanas, como veremos pronto). Sin embargo, los números de trabajo son más importantes: puedes usarlos con los comandos del shell para el *control de trabajos*.

Ya conoces la forma más obvia de controlar un trabajo: puedes crear uno en segundo plano con &. Una vez que un trabajo se está ejecutando en segundo plano, puedes dejar que se complete, llevarlo al *primer plano* o enviarle un mensaje llamado *signal*.

8.2.1. Primer plano y segundo plano

El comando integrado *fg* lleva un trabajo en segundo plano al primer plano. Normalmente, esto significa que el trabajo tiene control de tu terminal o ventana y, por lo tanto, puede aceptar tu entrada. En otras palabras, el trabajo comienza a actuar como si hubieras escrito su comando sin el &.

Si solo tienes un trabajo en segundo plano, puedes usar *fg* sin argumentos, y el shell lleva ese trabajo al primer plano. Pero si tienes varios trabajos en segundo plano, el shell elige el que pusiste en segundo plano más recientemente. Si deseas que se lleve al primer plano un

trabajo diferente, debes usar el nombre del comando del trabajo, precedido por un signo de porcentaje (%), o puedes usar su número de trabajo, también precedido por %, o su identificador de proceso sin un signo de porcentaje. Si no recuerdas qué trabajos se están ejecutando, puedes usar el comando *jobs* para listarlos.

Algunos ejemplos deberían aclarar esto. Digamos que creaste tres trabajos en segundo plano como se muestra arriba. Si escribes *jobs*, verás esto:

```
[1] Running     fred &
[2] - Running   bob &
[3] + Running   dave | george &
```

jobs tiene algunas opciones interesantes. Además del estado del trabajo, `jobs -l` también lista los identificadores de grupo de procesos:

```
[1]   2349    Running   fred &
[2] - 2367    Running   bob &
[3] + 2382    Running   dave | george &
```

¿Cómo funciona todo esto? Cada vez que ejecutas un trabajo, el o los procesos en el trabajo se colocan en un nuevo *grupo de procesos*. Cada proceso en un grupo de procesos, además de su número único de identificación de proceso, también tiene un *identificador de grupo de procesos(ID)*. El identificador de grupo de procesos es igual al identificador de proceso del *líder* del grupo de procesos, que es uno de los procesos invocados como parte del trabajo. (De hecho, el último en la tubería). Los números que imprime el shell son, de hecho, los identificadores de grupo de procesos. (Observa que para el trabajo 3, hay dos procesos, pero solo un número).

Ahora bien, tu dispositivo terminal, ya sea un puerto serie real o un seudoterminal como el que obtienes en un sistema de ventanas o una *sesión de telnet*, también tiene un número de identificación de grupo de procesos. Los procesos cuyo identificador de grupo de procesos coincide con el del terminal «poseen» el terminal, en el sentido de que se les permite leer la entrada desde él. En resumen, el control de trabajos funciona configurando el grupo de procesos del terminal para que sea el mismo que el grupo de procesos del trabajo actual. (Hay muchos más detalles técnicos, incluida la idea de una «sesión» introducida por POSIX, pero esos detalles no son necesarios para entender el uso cotidiano del control de trabajos).

La opción `-p` le dice a *jobs* que liste *solo* los identificadores de grupo de procesos:

```
$ jobs -p
2349
2367
2382
```

Esto podría ser útil con la sustitución de comandos; consulta la Tarea 8-1 más adelante en este capítulo. Finalmente, la opción `-n` lista solo aquellos trabajos cuyo estado ha cambiado desde la última vez que el shell lo informó, ya sea con un comando *jobs* o de otra manera.

Si escribes *fg* sin un argumento, el shell coloca a `dave | george` en primer plano, porque fue colocado en segundo plano más recientemente. Pero si escribes `fg %bob` (o `fg %2`), bob irá al primer plano.

También puedes referirte al trabajo colocado más recientemente en segundo plano con `%+`. De manera similar, `%-` se refiere al trabajo en segundo plano invocado a continuación más recientemente (*bob* en este caso). Eso explica los signos más y menos en lo anterior: el signo más muestra el trabajo invocado más recientemente; el signo menos muestra el trabajo invocado a continuación más recientemente. [2]

Si más de un trabajo en segundo plano tiene el mismo comando, entonces `%command` desambiguará eligiendo el trabajo invocado más recientemente (como cabría esperar). Si esto no es lo que deseas, debes usar el número del trabajo en lugar del nombre del comando. Sin embargo, si los comandos tienen argumentos diferentes, puedes usar `%?string` en lugar de `%command`. `%?string` se refiere al trabajo cuyo comando contiene la cadena. Por ejemplo, supongamos que iniciaste estos trabajos en segundo plano:

```
$ bob pete &
[1]   189
$ bob ralph &
[2]   190
$
```

Luego puedes usar `%?pete` y `%?ralph` para referirte a cada uno de ellos, aunque en realidad `%?pe` y `%?ra` son suficientes para desambiguar.

La Tabla 8.1 enumera todas las formas de referirse a trabajos en segundo plano. Hemos descubierto que, dado lo infrecuentemente que las personas utilizan comandos de control de trabajos, los números de trabajo o los nombres de comando son suficientes, y las demás formas son superfluas.

[2] Esto es análogo a ~+ y ~- como referencias al directorio actual y anterior; consulta la nota al pie en el CCapítulo 7. Además: %% es un sinónimo de %+.

Tabla 8.1: Formas de referirse a trabajos en segundo plano

Referencia	**Trabajo en segundo plano**
N	Identificación de proceso N
-N	Identificación de grupo de procesos N
%N	Número de trabajo N
%*string*	Trabajo cuyo comando comienza con *string*
%?*string*	Trabajo cuyo comando contiene *string*
%+, %%	Trabajo en segundo plano invocado más recientemente
%-	Segundo trabajo en segundo plano invocado más recientemente

8.2.2. Suspendiendo un trabajo

Al igual que puedes llevar trabajos en segundo plano al primer plano con *fg*, también puedes enviar un trabajo en primer plano al segundo plano. Esto implica suspender el trabajo para que el shell recupere el control de tu terminal.

Para suspender un trabajo, escribe CTRL-Z [3] mientras se está ejecutando. Esto es análogo a escribir CTRL-C (o la tecla de interrupción que tengas configurada), con la diferencia de que puedes reanudar el trabajo después de detenerlo. Cuando escribes CTRL-Z, el shell responde con un mensaje como este:

```
[1] + Detenido            comando
```

Luego te devuelve tu indicador de comando. También coloca el trabajo suspendido en la parte superior de la lista de trabajos, como indica el signo +.

Para reanudar un trabajo suspendido para que continúe ejecutándose en primer plano, simplemente escribe *fg*. Si, por alguna razón, pusiste otros trabajos en segundo plano después de escribir CTRL-Z, usa *fg* con un nombre o número de trabajo. Por ejemplo:

```
fred se está ejecutando...
CTRL-Z
[1] + Detenido          fred
$ bob &
[2] bob &
$ fg%fred
fred se reanuda en primer plano...
```

[3] Esto asume que la tecla CTRL-Z está configurada como tu tecla de suspensión; al igual que con CTRL-C e interrupciones, esto es convencional pero no obligatorio.

La capacidad de suspender trabajos y reanudarlos en primer plano resulta muy útil cuando solo tienes una conexión a tu sistema, [4] y estás utilizando un editor de texto como vi en un archivo que necesita ser procesado. Por ejemplo, si estás editando un archivo HTML para tu servidor web, puedes hacer lo siguiente:

```
$ vi myfile.html
Edita el archivo... CTRL-Z
[1] + Detenido          vi myfile.html
$ lynx myfile.html      Previsualiza los resultados con un navegador de solo texto
Ves que cometiste un error
$ fg
vi vuelve a abrir en el mismo lugar de tu archivo
```

Los programadores a menudo utilizan la misma técnica al depurar código fuente.

También es probable que encuentres útil suspender un trabajo y luego reanudarlo en segundo plano en lugar de en primer plano. Puedes iniciar un comando en primer plano (es decir, normalmente) y descubrir que tarda mucho más de lo esperado, por ejemplo, una búsqueda con *grep*, *sort* o una consulta a una base de datos. Necesitas que el comando finalice, pero también te gustaría recuperar el control de tu terminal para poder hacer otras tareas. Si escribes CTRL-Z seguido de *bg*, mueves el trabajo al segundo plano. [5]

8.2.3. Desvinculando un trabajo

Normalmente, cuando cierras la sesión, el shell envía la señal *HUP* (consulta la siguiente sección) a cualquier trabajo en segundo plano. Si has iniciado un trabajo de larga duración en segundo plano y deseas que se complete sin importar qué, debes indicárselo al shell mediante el comando *disown* con uno o más números de identificación de trabajo como argumentos. Sin argumentos, se desvinculan *todos* los trabajos en segundo plano.

8.3. Señales

Dijimos anteriormente que escribir CTRL-Z para suspender un trabajo es similar a escribir CTRL-C para detener un trabajo, excepto que puedes reanudar el trabajo más tarde. De hecho, son similares de una manera más profunda: ambos son casos particulares del acto de enviar una *señal* a un proceso.

[4] Como cuando te conectas desde casa a tu oficina, o estás conectado a un sistema remoto a través de Internet mediante *telnet* o *ssh*.

[5] Sin embargo, ten cuidado, ya que no todos los comandos se comportan de manera «adecuada» cuando haces esto. Ten especial precaución con los comandos que se ejecutan sobre una red en una máquina remota; podrías «confundir» al programa remoto.

Una señal es un mensaje que un proceso envía a otro cuando ocurre algún evento anormal o cuando quiere que el otro proceso haga algo. La mayoría de las veces, un proceso envía una señal a un subproceso que creó. Sin duda, ya te sientes cómodo con la idea de que un proceso puede comunicarse con otro a través de una tubería de entrada/salida; piensa en una señal como otra forma para que los procesos se comuniquen entre sí. (De hecho, cualquier libro de sistemas operativos te dirá que ambos son ejemplos del concepto general de *comunicación entre procesos*, o IPC por sus siglas en inglés). [6]

Dependiendo de la versión de Unix, hay entre dos y tres docenas de tipos de señales, incluyendo algunas que un programador puede usar para cualquier propósito. Las señales tienen números (del 1 al número de señales que el sistema admite) y nombres; nosotros usaremos los últimos. Puedes obtener una lista de todas las señales en tu sistema escribiendo `kill -l`. Ten en cuenta que, al escribir código de shell que involucre señales, los nombres de las señales son más portátiles a otras versiones de Unix que los números de señal.

8.3.1. Señales con Control-Key

Cuando escribes CTRL-C, le estás indicando al shell que envíe la señal INT (de «interrupción») al trabajo actual; CTRL-Z envía la señal TSTP (de «parada de terminal»). También puedes enviar al trabajo actual una señal QUIT escribiendo CTRL-\ (control barra invertida); esto es como una versión «más fuerte» de CTRL-C. [7] Normalmente usarías CTRL-\ cuando (y solo cuando) CTRL-C no funciona.

Como veremos pronto, también hay una señal de «pánico» llamada KILL que puedes enviar a un proceso cuando ni siquiera CTRL-\ funciona. Pero no está asociada a ninguna tecla de control, lo que significa que no puedes usarla para detener el proceso que se está ejecutando actualmente. INT, TSTP y QUIT son las únicas señales que puedes usar con teclas de control (aunque algunos sistemas tienen señales de teclas de control adicionales).

Puedes personalizar las teclas de control que se utilizan para enviar señales con opciones del comando *stty(1)*. Estas opciones varían de un sistema a otro; consulta la página de

[6] Las tuberías (pipes) y las señales eran los únicos mecanismos IPC en las primeras versiones de Unix. Las versiones más modernas tienen mecanismos adicionales, como sockets, tuberías con nombre y memoria compartida. Las tuberías con nombre son accesibles para los programadores de shell a través del comando *mkfifo(1)*, lo cual está más allá del alcance de este libro.

[7] CTRL-\ también puede hacer que el programa en ejecución deje un archivo llamado core en el directorio actual del programa. Este archivo contiene una imagen del proceso al cual le enviaste la señal; un programador podría usarlo para ayudar a depurar el programa que se estaba ejecutando. El nombre del archivo es un término (muy) anticuado para la memoria de una computadora. Otras señales también dejan estos «volcados de núcleo»; puedes eliminarlos a menos que un programador del sistema te diga lo contrario.

manual del comando para obtener información, pero la sintaxis habitual es `stty signame char`. *signame* es un nombre para la señal que, lamentablemente, a menudo no es el mismo que los nombres que usamos aquí. La Tabla 1.7 en el Capítulo 1 enumera los nombres *stty* para señales y las acciones del controlador de terminal que se encuentran en todas las versiones modernas de Unix. *char* es el carácter de control, que puedes dar en la misma notación que usamos. Por ejemplo, para configurar tu tecla INT en CTRL-X en la mayoría de los sistemas, usa:

```
stty intr ^X
```

Ahora que te hemos dicho cómo hacer esto, deberíamos agregar que no lo recomendamos. Cambiar las teclas de señal podría causar problemas si alguien más tiene que detener un proceso descontrolado en tu máquina.

La mayoría de las otras señales son utilizadas por el sistema operativo para informar a los procesos sobre condiciones de error, como una instrucción de código de máquina incorrecta, una dirección de memoria incorrecta, división por cero u otros eventos como la disponibilidad de entrada en un descriptor de archivo o el temporizador («alarma» en la terminología de Unix) que se activa. Las señales restantes se utilizan para condiciones de error esotéricas que solo interesan a los programadores de sistemas de bajo nivel; las versiones más nuevas de Unix tienen tipos de señales cada vez más arcanos.

8.3.2. kill

Puedes utilizar el comando incorporado del shell, *kill*, para enviar una señal a cualquier proceso que hayas creado, no solo al trabajo que se está ejecutando actualmente. *kill* toma como argumento el ID del proceso, el número de trabajo o el nombre del comando del proceso al cual deseas enviar la señal. Por defecto, *kill* envía la señal TERM («terminar»), que generalmente tiene el mismo efecto que la señal INT que envías con CTRL-C. Pero puedes especificar una señal diferente utilizando la opción -s y el nombre de la señal, o la opción -n y un número de señal.

kill recibe su nombre debido a la naturaleza de la señal TERM predeterminada, pero hay otra razón, que tiene que ver con la forma en que Unix maneja las señales en general. Los detalles completos son demasiado complejos para entrar en ellos aquí, pero la siguiente explicación debería ser suficiente.

La mayoría de las señales hacen que un proceso que las recibe se detenga y muera; por lo tanto, si envías cualquiera de estas señales, «matas» el proceso que la recibe. Sin embargo,

los programas pueden configurarse para «atrapar» señales específicas y tomar alguna otra acción. Por ejemplo, un editor de texto haría bien en guardar el archivo que se está editando antes de terminar cuando recibe una señal como INT, TERM o QUIT. Determinar qué hacer cuando llegan diversas señales es parte de la diversión de la programación de sistemas Unix.

Aquí tienes un ejemplo de *kill*. Supongamos que tienes un proceso *fred* en segundo plano, con un ID de proceso 480 y un número de trabajo 1, que necesita detenerse. Comenzarías con este comando:

```
kill %1
```

Si tuviste éxito, verías un mensaje como este:

```
[1] + Terminated      fred &
```

Si no ves esto, entonces la señal TERM no logró detener el trabajo. El siguiente paso sería intentar con QUIT:

```
kill -s QUIT %1
```

Si eso funcionó, verías este mensaje:

```
[1] + Quit(coredump)      fred &
```

El shell indica la señal que mató al programa (*Quit*) y el hecho de que produjo un archivo de núcleo. Cuando un programa sale de manera normal, el estado de salida que devuelve al shell es un valor entre 0 y 255. Cuando un programa muere al recibir una señal, sale no con un valor de estado de su elección, sino con el estado 256+N, donde N es el número de la señal que recibió. (Con *ksh88* y la mayoría de las otras shells, los estados de salida normales están entre 0 y 127, y el estado de «muerte por señal» es 128+N. Caveat emptor.)

Si ni siquiera `QUIT` funciona, el método de último recurso sería usar `KILL`:

```
kill -s KILL %1
```

(Observa cómo esto tiene el sabor de «gritarle» al proceso fuera de control). Esto produce el mensaje:

```
[1] + Killed      fred &
```

Es imposible que un proceso atrape una señal KILL: el sistema operativo debería terminar el proceso de inmediato e incondicionalmente. Si no lo hace, entonces tu proceso está en uno de los «estados divertidos» que veremos más adelante en este capítulo, o (mucho menos probable) hay un error en tu versión de Unix.

En sistemas de control de trabajos, hay una señal adicional e inatrapable: STOP. Esto es similar a TSTP, ya que suspende el trabajo objetivo. Pero a diferencia de TSTP, no se puede capturar ni ignorar. Es una señal más drástica que TSTP, pero menos que QUIT o TERM, ya que un trabajo detenido aún se puede continuar con *fg* o *bg*. El shell Korn proporciona el alias predefinido `stop='kill -s STOP'` para facilitar la detención de trabajos.

La Tarea 8-1 es otro ejemplo de cómo usar el comando *kill*.

Tarea 8-1

Escribe una función llamada *killalljobs* que mate todos los trabajos en segundo plano.[a]

[a] Para probar tu comprensión de cómo funciona el shell, responde a esta pregunta: ¿por qué esto no se puede hacer como un script separado?

La solución para esta tarea es simple, utilizando `jobs -p`:

```
function killalljobs {
  kill "$@" $(jobs -p)
}
```

Es posible que te sientas tentado a usar la señal KILL inmediatamente, en lugar de probar primero TERM (la predeterminada) y QUIT. No lo hagas. TERM y QUIT están diseñadas para darle al proceso la oportunidad de limpiar antes de salir, mientras que KILL detendrá el proceso, sin importar en qué parte de su cálculo se encuentre. *Usa KILL solo como último recurso.*

Puedes usar el comando *kill* con cualquier proceso que crees, no solo trabajos en segundo plano de tu shell actual. Por ejemplo, si usas un sistema de ventanas, es posible que tengas varias ventanas de terminal, cada una de las cuales ejecuta su propia shell. Si una shell está ejecutando un proceso que deseas detener, puedes matarlo desde otra ventana, pero no puedes referirte a él con un número de trabajo porque se está ejecutando bajo una shell diferente. Debes usar su ID de proceso en su lugar.

8.3.3. ps

Esta es probablemente la única situación en la que un usuario casual necesitaría conocer el ID de un proceso. El comando *ps(1)* te brinda esta información; sin embargo, también puede darte mucha información adicional por la cual debes navegar.

ps es un comando complejo. Tiene muchas opciones, algunas de las cuales difieren de una versión de Unix a otra. Para agregar a la confusión, es posible que necesites opciones

diferentes en diferentes versiones de Unix para obtener la misma información. Usaremos opciones disponibles en los dos principales tipos de sistemas Unix, los derivados de System V (como la mayoría de las versiones para Intel x86, así como Solaris, AIX de IBM y HP-UX de Hewlett-Packard) y BSD (Ultrix de Compaq, SunOS 4.x y también GNU/Linux). Si no estás seguro de qué tipo de versión de Unix tienes, prueba primero con las opciones de System V.

Puedes invocar *ps* en su forma más simple sin ninguna opción. En este caso, imprime una línea de información sobre el shell de inicio de sesión actual y cualquier proceso que se esté ejecutando bajo ella (es decir, trabajos en segundo plano). Por ejemplo, si invocaste tres trabajos en segundo plano, como vimos anteriormente en el capítulo, *ps* en versiones derivadas de System V de Unix produciría una salida que se parece a esto:

```
PID   TTY       TIME  COMD
146   pts/10    0:03  ksh
2349  pts/10    0:03  fred
2367  pts/10    0:17  bob
2387  pts/10    0:06  george
2389  pts/10    0:09  dave
2390  pts/10    0:00  ps
```

La salida en sistemas derivados de BSD se vería así:

```
PID   TT    STAT      TIME  COMMAND
146   10    S         0:03  /bin/ksh -i
2349  10    R         0:03  fred
2367  10    D         0:17  bob
2387  10    S         0:06  george
2389  10    R         0:09  dave
2390  10    R         0:00  ps
```

(Puedes ignorar la columna STAT). Esto es un poco como el comando *jobs*. PID es el ID del proceso; TTY (o TT) es el terminal (o pseudo-terminal, si estás utilizando un sistema de ventanas) desde el cual se invocó el proceso; TIME es la cantidad de tiempo del procesador (no el tiempo real o «de reloj») que el proceso ha utilizado hasta ahora; COMD (o COMMAND) es el comando. Observa que la versión BSD incluye los argumentos del comando, si los hay; también observa que la primera línea informa sobre el proceso del shell principal, y en la última línea, *ps* informa sobre sí mismo.

ps sin argumentos lista todos los procesos iniciados desde el terminal o pseudo-terminal actual. Pero como *ps* no es un comando de shell, no correlaciona los ID de procesos con los números de trabajo del shell. Tampoco te ayuda a encontrar el ID del proceso fuera de control en otra ventana del shell.

Para obtener esta información, usa `ps -a` (para «todo»); esto lista información sobre un conjunto diferente de procesos, dependiendo de tu versión de Unix.

System V

En lugar de enumerar todos los procesos que se iniciaron bajo un terminal específico, `ps -a` en sistemas derivados de System V lista todos los procesos asociados con cualquier terminal que no sean líderes de grupo. Para nuestros propósitos, un «líder de grupo» es el shell principal de un terminal o ventana. Por lo tanto, si estás utilizando un sistema de ventanas, `ps -a` lista todos los trabajos iniciados en todas las ventanas (por todos los usuarios), pero no sus shells principales.

Supongamos que, en el ejemplo anterior, tienes solo un terminal o ventana. Entonces, `ps -a` imprime la misma salida que *ps* simple, excepto por la primera línea, ya que esa es el shell principal. Esto no parece ser muy útil.

Pero considera lo que sucede cuando tienes varias ventanas abiertas. Digamos que tienes tres ventanas, todas ejecutando emuladores de terminal como *xterm* para el Sistema de Ventanas X. Inicias trabajos en segundo plano *fred, dave* y *bob* en ventanas con números de pseudo-terminal 1, 2 y 3, respectivamente. Esta situación se muestra en la Figura 8.1.

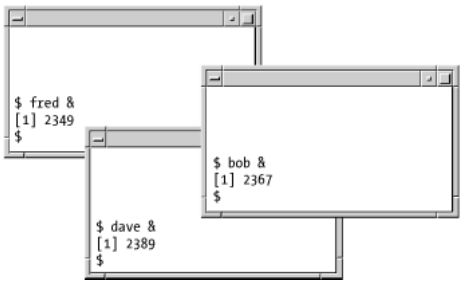

Figura 8.1: Trabajos en segundo plano en múltiples ventanas

Supongamos que estás en la ventana superior. Si escribes *ps*, verás algo como esto:

```
PID   TTY      TIME  COMD
146   pts/1    0:03  ksh
2349  pts/1    0:03  fred
2390  pts/1    0:00  ps
```

Pero si escribes *ps -a*, verás esto:

```
PID    TTY      TIME  COMD
2349   pts/1    0:03  fred
2367   pts/2    0:17  bob
2389   pts/3    0:09  dave
2390   pts/1    0:00  ps
```

Ahora deberías ver cómo `ps -a` puede ayudarte a rastrear un proceso fuera de control. Si es *dave*, puedes escribir `kill 2389`. Si eso no funciona, intenta `kill -s QUIT 2389`, o en el peor caso, `kill -s KILL 2389`.

BSD

En sistemas derivados de BSD, [8] *ps -a* lista todos los trabajos que se iniciaron en cualquier terminal; en otras palabras, es un poco como concatenar los resultados de *ps* simple para cada usuario en el sistema. Dado el escenario anterior, `ps -a` te mostrará todos los procesos que la versión de System V muestra, además de los líderes de grupo (shells principales).

Desafortunadamente, `ps -a` (en cualquier versión de Unix) no informará sobre procesos que se encuentren en ciertas condiciones patológicas donde «olvidan» cosas como qué shell los invocó y a qué terminal pertenecen. Estos procesos tienen nombres coloridos (zombies, huérfanos) que realmente se utilizan en la literatura técnica de Unix, no solo de manera informal por programadores de sistemas profesionales. Si tienes un problema grave con un proceso fuera de control, es posible que el proceso haya ingresado a uno de estos estados.

No nos preocupemos por el porqué o cómo un proceso llega a este estado. Todo lo que necesitas entender es que el proceso no aparecerá cuando escribas `ps -a`. Necesitas otra opción para *ps* para verlo: en System V, es `ps -e` («everything»), mientras que en BSD, es `ps -ax`.

Estas opciones indican a *ps* que liste procesos que no se iniciaron desde terminales o que «olvidaron» desde qué terminal se iniciaron. La primera categoría incluye muchos procesos que probablemente ni siquiera sabías que existían: estos incluyen procesos básicos que ejecutan el sistema y los llamados *daemons* (pronunciado <demonios») que manejan servicios del sistema como correo, impresión, sistemas de archivos en red, etc.

De hecho, la salida de `ps -e` o `ps -ax` es una excelente fuente de educación sobre los entresijos internos del sistema Unix, si tienes curiosidad por conocerlos. Ejecuta el comando en tu sistema y, para cada línea de la lista que parezca interesante, invoca ‘man‘ en el nombre del proceso o búscalo en el Manual del Programador de Unix para tu sistema.

Las shells de usuario y los procesos se enumeran en la parte inferior de la salida de `ps -e` o `ps -ax`; aquí es donde debes buscar procesos fuera de control. Observa que muchos procesos en la lista tienen un ? en lugar de un terminal. Estos bien no deberían tener uno (como los *daemons* básicos) o son procesos fuera de control. Por lo tanto, es probable que si `ps`

[8] En sistemas GNU/Linux, *ps* actúa como la versión de BSD.

`-a` no encuentra un proceso que estás intentando eliminar, `ps -e` (o `ps -ax`) lo mostrará con ? en la columna TTY (o TT). Puedes determinar qué proceso deseas observando la columna COMD (o COMMAND).

8.3.4. kill: La historia completa

El comando *kill* tiene un nombre bastante equívoco. Debería haberse llamado *sendsignal* o algo similar, ya que envía señales a los procesos. (De hecho, el nombre se deriva de la llamada al sistema *kill(2)*, que el comando *kill* utiliza para enviar señales, y que está igualmente mal nombrada).

Como vimos anteriormente, `kill -l` te proporciona la lista completa de nombres de señales disponibles en tu sistema. El comportamiento de la versión incorporada de *kill* se ha racionalizado considerablemente en *ksh93*. Las opciones y lo que hacen se resumen en la Tabla 8.2.

Tabla 8.2: Opciones para kill

Opción	**Significado**
kill *job...*	Envía la señal TERM a cada *trabajo* nombrado. Este es el uso normal.
kill -l	Enumera los nombres de todas las señales compatibles.
kill -l *signal...*	Si *signal* es un número, imprime su nombre. Si es un nombre, imprime su número. Si *signal* es un número mayor que 256, se trata como un estado de salida. El shell resta 256 e imprime la señal correspondiente.
kill -s *signal-name job...*	Envía la señal nombrada por *signal-name* a cada trabajo dado.
kill -n *signal-number job...*	Envía la señal numérica dada por *signal-number* a cada trabajo dado.
kill *-signal job...*	Envía la señal especificada por *signal* a cada *trabajo* dado. *signal* puede ser un número o un nombre de señal. Esta forma se considera obsoleta; se proporciona por compatibilidad con *ksh88* y la orden externa *kill(1)*.

Un lugar para aprovechar la capacidad de *kill* de convertir un número en un nombre es al emitir diagnósticos. Cuando un trabajo muere debido a una señal, el estado de salida es 256 más el número de señal. Por lo tanto, podrías usar un código como este para producir un diagnóstico significativo desde un script:

```
es=$?       # guarda el estado de salida
if ((es >= 256)); then
  print "el trabajo recibió la señal $(kill -l $((es - 256)))"
fi
```

8.4. trap

Hemos estado discutiendo cómo las señales afectan al usuario casual; ahora hablemos un poco sobre cómo los programadores de shell pueden usarlas. No profundizaremos demasiado en esto, porque realmente es el ámbito de los programadores de sistemas.

Mencionamos anteriormente que los programas en general pueden configurarse para «atrapar» señales específicas y procesarlas a su manera. El comando integrado *trap* te permite hacer esto desde un script de shell. *trap* es crucial para «blindar» programas de shell extensos para que reaccionen adecuadamente a eventos anómalos, al igual que los programas en cualquier lenguaje deberían protegerse contra una entrada no válida. También es importante para ciertas tareas de programación de sistemas, como veremos en el próximo capítulo.

La sintaxis de trap es:

```
trap cmd sig1 sig2 ...
```

Es decir, cuando se recibe cualquiera de *sig1, sig2*, etc., ejecuta *cmd* y luego continúa la ejecución. Después de que *cmd* finaliza, el script continúa la ejecución justo después del comando que fue interrumpido. [9]

Por supuesto, *cmd* puede ser un script o una función. Las señales pueden especificarse por nombre o por número. También puedes invocar *trap* sin argumentos, en cuyo caso el shell imprime una lista de cualquier trampa que se haya establecido, utilizando nombres simbólicos para las señales. Si usas `trap -p`, el shell imprime la configuración de la trampa de una manera que se puede guardar y volver a leer más tarde mediante una invocación diferente del shell.

El shell escanea el texto de *cmd* dos veces. La primera vez es mientras se prepara para ejecutar el comando *trap*; todas las sustituciones como se describen en el Capítulo 7 se realizan antes de ejecutar el comando *trap*. La segunda vez es cuando el shell ejecuta realmente la trampa. Por esta razón, es mejor usar comillas simples alrededor de *cmd* en el texto del programa de shell. Cuando el shell ejecuta el comando de la trampa, `$?` es siempre el estado de salida del último comando ejecutado antes de que comenzara la trampa. Esto es importante para diagnósticos.

[9] Esto es lo que *suele* suceder. A veces, el comando en ejecución actual se aborta (*sleep* actúa así, como veremos pronto); otras veces, termina su ejecución. Los detalles adicionales están más allá del alcance de este libro.

Aquí tienes un ejemplo simple que muestra cómo funciona *trap*. Supongamos que tenemos un script de shell llamado loop con este código:

```
while true; do
  sleep 60
done
```

Esto simplemente pausa durante 60 segundos (el comando *sleep(1)*) y se repite indefinidamente. *true* es un comando «sin hacer nada» cuyo estado de salida siempre es 0. Por eficiencia, está integrado en el shell. (El comando *false* es un comando similar «sin hacer nada» cuyo estado de salida siempre es 1. También está integrado en el shell.) Como sucede, *sleep* también está integrado en el shell. Intenta escribir este script. Invócalo, déjalo correr un rato y luego escribe CTRL-C (asumiendo que esa es tu tecla de interrupción). Debería detenerse y deberías recuperar tu indicador de shell.

Ahora inserta la siguiente línea al principio del script:

```
trap 'print "¡Presionaste control-C!"' INT
```

Invoca el script de nuevo. Ahora presiona CTRL-C. Las probabilidades son abrumadoras de que estés interrumpiendo el comando *sleep* (en lugar de *true*). Deberías ver el mensaje «¡Presionaste control-C!», y el script no se detendrá; en cambio, el comando *sleep* se abortará y volverá a empezar otro *sleep*. Presiona CTRL-\ para detenerlo. Escribe `rm core` para deshacerte del archivo de volcado de núcleo resultante.

A continuación, ejecuta el script en segundo plano escribiendo `loop &`. Escribe `kill %loop` (es decir, envíale la señal TERM); el script se detendrá. Agrega TERM al comando de trampa, de modo que se vea así:

```
trap 'print "¡Presionaste control-C!"' INT TERM
```

Ahora repite el proceso: ejecútalo en segundo plano y escribe `kill %loop`. Como antes, verás el mensaje y el proceso seguirá ejecutándose. Escribe `kill -KILL %loop` para detenerlo.

Observa que el mensaje no es realmente apropiado cuando usas *kill*. Cambiaremos el script para que imprima un mensaje mejor en el caso de *kill*:

```
trap 'print "¡Presionaste control-C!"' INT
trap 'print "¡Intentaste matarme!"' TERM
while true; do
  sleep 60
done
```

Ahora pruébalo de ambas maneras: en primer plano con CTRL-C y en segundo plano con *kill*. Verás mensajes diferentes.

8.4.1. Trampasy funciones

La relación entre las trampas y las funciones del shell es directa, pero tiene ciertos matices que vale la pena discutir. Lo más importante que debes entender es que las funciones del shell Korn (aquellas creadas con la palabra clave `function`; consulta el CCapítulo 4) tienen sus propias trampas locales; estas no son conocidas fuera de la función. Las funciones de estilo antiguo de POSIX (aquellas creadas con la sintaxis `nombre()`) comparten trampas con el script principal.

Comencemos con las funciones de estilo `function`, donde las trampas son locales. En particular, el script circundante no las conoce. Considera este código:

```
function settrap {
  trap 'print "You hit control-C!"' INT
}

settrap
while true; do
  sleep 60
done
```

Si invocas este script y presionas tu tecla de interrupción, simplemente se cierra. La trampa en INT en la función solo es conocida dentro de esa función. Por otro lado:

```
function loop {
  trap 'print "How dare you!"' INT
  while true; do
    sleep 60
  done
}

trap 'print "You hit control-C!"' INT
loop
```

Cuando ejecutas este script y presionas tu tecla de interrupción, imprime «¡Cómo te atreves!» Pero, ¿qué tal esto?

```
function loop {
  while true; do
  sleep 60
  done
}

trap 'print "You hit control-C!"' INT
loop
print 'exiting ...'
```

Esta vez, el código de bucle está dentro de una función, y la trampa se establece en el script

circundante. Si presionas tu tecla de interrupción, imprime el mensaje y luego imprime «saliendo...». No repite el bucle como antes.

¿Por qué? Recuerda que cuando la señal entra, el shell aborta el comando actual, que en este caso es una llamada a una función. Toda la función se aborta, y la ejecución se reanuda en la siguiente declaración después de la llamada a la función.

La ventaja de las trampas locales a las funciones es que te permiten controlar el comportamiento de una función por separado del código circundante.

Sin embargo, es posible que desees definir trampas globales dentro de funciones. Existe una forma bastante chapucera de hacerlo; depende de una función que presentamos en el Capítulo 9, que llamamos «falsa señal». Aquí hay una manera de configurar trapcode como una trampa global para la señal SIG dentro de una función:

```
trap "trap trapcode sig" EXIT
```

Esto configura el comando `trap trapcode SIG` para ejecutarse justo después de que la función salga, momento en el cual el script del shell circundante está en el ámbito (es decir, está «a cargo»). Cuando se ejecuta ese comando, *trapcode* se configura para manejar la señal SIG.

Por ejemplo, es posible que desees restablecer la trampa en la señal que acabas de recibir, así:

```
function trap_handler {
  trap "trap second_handler INT" EXIT
  print 'Interrupt: one more to abort.'
}

function second_handler {
  print 'Aborted.' exit
}

trap trap_handler INT
```

Este código actúa como la utilidad de correo Unix: cuando estás escribiendo un mensaje, debes presionar tu tecla de interrupción dos veces para abortar el proceso.

Hay una forma menos chapucera de hacer esto, aprovechando el hecho de que las funciones de estilo POSIX comparten trampas con el script principal:

```
# POSIX style function, trap is global
trap_handler () {
  trap second_handler INT
  print 'Interrupt: one more to abort.'
```

```
}

function second_handler {
  print 'Aborted.' exit
}

trap trap_handler INT

while true ; do
  sleep 60
done
```

Si escribes esto y lo ejecutas, obtendrás los mismos resultados que en el ejemplo anterior, sin la artimaña adicional de usar la falsa señal EXIT.

Hablando de *mail*, en la Tarea 8-2 mostraremos un ejemplo más práctico de trampas.

Tarea 8-2

Como parte de un sistema de correo electrónico, escribe el código de shell que permite a un usuario componer un mensaje.

La idea básica es usar *cat* para crear el mensaje en un archivo temporal y luego pasar el nombre del archivo a un programa que realmente envía el mensaje a su destino. El código para crear el archivo es muy simple:

```
msgfile=/tmp/msg$$
cat > $msgfile
```

Dado que *cat* sin un argumento lee desde la entrada estándar, esto simplemente espera a que el usuario escriba un mensaje y lo termine con el carácter de fin de archivo CTRL-D

8.4.2. Variables de ID de proceso y ficheros temporales

Lo único nuevo aquí es $$ en la expresión del nombre de archivo. Esta es una variable especial del shell cuyo valor es el ID de proceso del shell actual.

Para ver cómo funciona $$, escribe `ps` y toma nota del ID de proceso de tu proceso de shell (*ksh*). Luego, escribe `print "$$"`; el shell responderá con ese mismo número. Ahora escribe `ksh` para iniciar un subproceso del shell y, cuando obtengas un indicador, repite el proceso. Deberías ver un número diferente, probablemente ligeramente superior al último.

Puedes examinar la relación entre padres e hijos con más detalle utilizando la variable `PPID` (ID de proceso padre). *ksh* establece esto como el ID de proceso del proceso padre.

Cada vez que inicias una nueva instancia de *ksh*, si escribes `print $PPID`, deberías ver un número que es igual a $$ del shell anterior.

Otra variable del shell relacionada es *!* (es decir, su valor es `$!`), que contiene el ID de proceso del trabajo en segundo plano más recientemente invocado. Para ver cómo funciona esto, inicia cualquier trabajo en segundo plano y toma nota del ID de proceso impreso por el shell junto a `[1]`. Luego escribe `print "$!"`; deberías ver el mismo número.

Volviendo a nuestro ejemplo de correo: dado que todos los procesos en el sistema deben tener ID de procesos únicos, $$ es excelente para construir nombres de archivos temporales. Vimos un ejemplo de esto en el Capítulo 7, cuando discutíamos los pasos de evaluación de la línea de comandos, y también hay ejemplos en el Capítulo 9. [10]

El directorio `/tmp` se utiliza convencionalmente para archivos temporales. Por lo general, los archivos en este directorio se borran cada vez que se reinicia la computadora.

Sin embargo, un programa debería limpiar esos archivos antes de salir para evitar ocupar espacio en disco innecesario. Podríamos hacer esto en nuestro código muy fácilmente agregando la línea `rm $msgfile` después del código que realmente envía el mensaje. Pero, ¿qué pasa si el programa recibe una señal durante la ejecución? Por ejemplo, ¿qué pasa si un usuario cambia de opinión sobre el envío del mensaje y presiona CTRL-C para detener el proceso? Necesitaríamos limpiar antes de salir. Emularemos el sistema Unix real guardando el mensaje que se está escribiendo en un archivo llamado *dead.letter* en el directorio actual. Podemos hacer esto usando *trap* con una cadena de comandos que incluye un comando de salida:

```
trap 'mv $msgfile dead.letter; exit' INT TERM
msgfile=/tmp/msg$$
cat > $msgfile
# send the contents of $msgfile to the specified mail address ...
rm $msgfile
```

Cuando el script recibe una señal INT o TERM, guarda el archivo temporal y luego sale. Ten en cuenta que la cadena de comandos no se evalúa hasta que necesita ejecutarse, por lo que `$msgfile` contendrá el valor correcto; es por eso que rodeamos la cadena con comillas simples.

Pero, ¿qué pasa si el script recibe una señal antes de que se cree `msgfile` - aunque sea poco probable? En ese caso, *mv* intentará cambiar el nombre de un archivo que no existe. Para

[10] En la práctica, los nombres de archivos temporales basados solo en $$ pueden llevar a sistemas inseguros. Si tienes el programa *mktemp(1)* en tu sistema, deberías usarlo en tus aplicaciones para generar nombres únicos para tus archivos temporales.

solucionar esto, necesitamos verificar la existencia del archivo `$msgfile` antes de intentar guardarlo. El código para esto es un poco incómodo para ponerlo en una sola cadena de comandos, así que usaremos una función en su lugar:

```
function cleanup {
  if [[ -e $msgfile ]]; then
    mv $msgfile dead.letter
  fi
  exit
}

trap cleanup INT TERM
msgfile=/tmp/msg$$
cat > $msgfile
# send the contents of $msgfile to the specified mail address ...
rm $msgfile
```

8.4.3. Ignorando señales

A veces, llega una señal que no deseas manejar. Si proporcionas la cadena nula (o `''`) como argumento de comando para *trap*, el shell efectivamente ignora esa señal. El ejemplo clásico de una señal que es posible que desees ignorar es *HUP* (hangup), la señal que reciben todos tus procesos en segundo plano cuando cierras sesión. (Si tu conexión realmente se cae, Unix envía la señal *HUP* al shell. El shell reenvía la señal a todos tus procesos en segundo plano o la envía por iniciativa propia si cierras sesión de forma normal).

HUP tiene el comportamiento predeterminado habitual: mata al proceso que la recibe. Pero seguramente habrá momentos en los que no querrás que un trabajo en segundo plano finalice cuando cierras sesión. Por ejemplo, podrías iniciar una compilación larga o un trabajo de formato de texto; quieres cerrar sesión y volver más tarde cuando esperas que el trabajo esté terminado. En circunstancias normales, tu trabajo en segundo plano terminaría cuando cierras sesión. Pero si lo ejecutas en un entorno de shell donde la señal *HUP* se ignora, el trabajo se completa.

Para hacer esto, podrías escribir una función simple que se ve así:

```
function ignorehup {
  trap "" HUP
  eval "$@"
}
```

Escribimos esto como una función en lugar de un script por razones que se harán más claras cuando examinemos detenidamente los subprocesos al final de este capítulo.

En realidad, hay un comando Unix llamado *nohup* que hace precisamente esto. La función *start* del último capítulo podría incluir *nohup*:

```
function start {
  eval nohup "$@" > logfile 2>&1 &
}
```

Esto evita que *HUP* termine tu comando y guarda su salida estándar y de error en un archivo. De hecho, lo siguiente es igualmente bueno:

```
function start {
  nohup "$@" > logfile 2>&1 &
}
```

Si comprendes por qué *eval* es prácticamente redundante cuando usas *nohup* en este caso, entonces tienes un sólido entendimiento del material en el Capítulo 7.

8.4.4. Restablecimiento de trampas

Otro «caso especial» del comando *trap* ocurre cuando proporcionas un guion (-) como argumento de comando. Esto restablece la acción tomada cuando se recibe la señal al valor predeterminado, que generalmente es la terminación del proceso.

Como ejemplo de esto, volvamos a la Tarea 8-2, nuestro programa de correo. Después de que el usuario ha terminado de enviar el mensaje, el archivo temporal se borra. En ese momento, ya que ya no hay necesidad de «limpiar», podemos restablecer la trampa de señal a su estado predeterminado. El código para esto, aparte de las definiciones de funciones, es:

```
trap cleanup INT TERM

msgfile=/tmp/msg$$
cat > $msgfile
# enviar el contenido de $msgfile a la dirección de correo especificada ...
rm $msgfile

trap - INT TERM
```

La última línea de este código restablece los manejadores para las señales INT y TERM.

En este punto, es posible que estés pensando que uno podría involucrarse seriamente con el manejo de señales en un script de shell. Es cierto que los programas de alta resistencia dedican cantidades considerable de código al manejo de señales. Pero estos programas son casi siempre lo suficientemente grandes como para que el código de manejo de señales sea

una fracción muy pequeña del conjunto. Por ejemplo, puedes apostar a que el sistema de correo real de Unix es bastante resistente.

Sin embargo, es probable que nunca escribas un script de shell lo suficientemente complejo y que necesite ser lo suficientemente robusto como para justificar mucho manejo de señales. Puedes escribir un prototipo para un programa tan grande como el correo en código de shell, pero los prototipos, por definición, no necesitan ser a prueba de balas.

Por lo tanto, no deberías preocuparte por poner código de manejo de señales en cada script de shell de 20 líneas que escribas. Nuestro consejo es determinar si hay situaciones en las que una señal podría hacer que tu programa haga algo seriamente malo y agregar código para manejar esas contingencias. ¿Qué es «seriamente malo»? Bueno, con respecto a los ejemplos anteriores, diríamos que el caso en el que *HUP* hace que tu trabajo termine al cerrar la sesión es seriamente malo, mientras que la situación del archivo temporal en nuestro programa de correo no lo es.

El shell Korn tiene varias opciones nuevas para *trap* (con respecto al mismo comando en la mayoría de los shell de Bourne) que lo hacen útil como ayuda para depurar scripts de shell. Las cubrimos en el Capítulo 9.

8.5. Corrutinas

Hemos dedicado las últimas páginas a detalles casi microscópicos del comportamiento de los procesos. En lugar de continuar nuestra inmersión en las profundidades turbias, volveremos a una vista de más alto nivel de los procesos.

Anteriormente en este capítulo, cubrimos formas de controlar múltiples trabajos simultáneos dentro de una sesión interactiva de inicio de sesión; ahora consideramos el control de múltiples procesos dentro de programas de shell. Cuando dos (o más) procesos están programados explícitamente para ejecutarse simultáneamente y posiblemente comunicarse entre sí, los llamamos corrutinas.

Esto no es realmente nuevo: una *tubería* es un ejemplo de corrutinas. El constructo de una tubería del shell encapsula un conjunto bastante sofisticado de reglas sobre cómo interactúan los procesos entre sí. Si observamos más de cerca estas reglas, podremos entender mejor otras formas de manejar corrutinas, la mayoría de las cuales resultan ser más simples que las tuberías.

Cuando invocas una tubería simple, por ejemplo, `ls | more`, el shell invoca una serie de operaciones primitivas de Unix, también conocidas como *llamadas al sistema*. En efecto, el shell le indica a Unix que haga lo siguiente; en caso de que estés interesado, incluimos entre paréntesis la llamada al sistema real utilizada en cada paso:

1. Crear la tubería que manejará la entrada/salida entre los procesos (llamada al sistema *pipe*).
2. Crear dos subprocesos, que llamaremos P1 y P2 (*fork*).
3. Configurar la entrada/salida entre los procesos para que la salida estándar de P1 alimente la entrada estándar de P2 (*dup*, *close*).
4. Iniciar `/bin/ls` en el proceso P1 (*exec*).
5. Iniciar `/bin/more` en el proceso P2 (*exec*).
6. Esperar a que ambos procesos terminen (*wait*).

Probablemente puedas imaginar cómo cambian los pasos anteriores cuando la tubería involucra más de dos procesos.

Ahora simplifiquemos las cosas. Veremos cómo hacer que varios procesos se ejecuten al mismo tiempo si los procesos no necesitan comunicarse. Por ejemplo, queremos que los procesos *dave* y *bob* se ejecuten como corrutinas, sin comunicación, en un script de shell. Ambos deben ejecutarse por completo antes de que el script finalice. Nuestra solución inicial sería esta:

```
dave &
bob
```

Supongamos por un momento que *bob* es el último comando en el script. Lo anterior funciona, pero solo si *dave* termina primero. Si *dave* aún se está ejecutando cuando el script termina, se convierte en un proceso *huérfano*, es decir, entra en uno de los «estados extraños» que mencionamos anteriormente en este capítulo. No importa los detalles de ser un huérfano; simplemente cree que no quieres que esto suceda, y si ocurre, es posible que necesites usar el método de «proceso fugitivo» para detenerlo, discutido anteriormente en este capítulo. (Por ejemplo, considera el caso en el que *dave* se lanza a un consumo excesivo de recursos, ralentizando tu sistema, y es mucho más difícil detenerlo si el script principal ya ha salido).

8.5.1. wait

Hay una manera de asegurarse de que el script no termine antes que *dave*: el comando integrado *wait*. Sin argumentos, *wait* simplemente espera hasta que todos los trabajos en segundo plano hayan terminado. Entonces, para asegurarse de que el código anterior se comporte correctamente, agregaríamos *wait* de la siguiente manera:

```
dave &
bob
wait
```

Aquí, si *bob* termina primero, el shell principal espera a que *dave* termine antes de finalizar.

Si tu script tiene más de un trabajo en segundo plano y necesitas esperar a que terminen trabajos específicos, puedes darle a *wait* el mismo tipo de argumento de trabajo (con un signo de porcentaje) que usarías con *kill, fg* o *bg*.

Sin embargo, es probable que encuentres que *wait* sin argumentos es suficiente para todas las corrutinas que programarás. Las situaciones en las que necesitarías esperar trabajos en segundo plano específicos son bastante complejas y están fuera del alcance de este libro.

8.5.2. Ventajas y desventajas de las corrutinas

De hecho, puede que te preguntes por qué necesitarías programar corrutinas que no se comunican entre sí. Por ejemplo, ¿por qué no simplemente ejecutar *bob* después de *dave* de la manera habitual? ¿Qué ventaja hay en ejecutar los dos trabajos simultáneamente?

Si estás ejecutando un equipo con un solo procesador (CPU), hay una ventaja de rendimiento, pero solo si tienes desactivada la opción *bgnice* (ver Capítulo 3), y aún así solo en ciertas situaciones.

Hablando en términos generales, puedes caracterizar un proceso en función de cómo utiliza los recursos del sistema de tres maneras: si es *intensivo en CPU* (por ejemplo, realiza muchos cálculos numéricos), *intensivo en E/S* (realiza mucha lectura o escritura en el disco) o *interactivo* (requiere intervención del usuario).

Ya sabemos desde el Capítulo 1 que no tiene sentido ejecutar un trabajo interactivo en segundo plano. Pero aparte de eso, cuanto más difieran dos o más procesos en estos tres criterios, mejor es ejecutarlos simultáneamente. Por ejemplo, un cálculo estadístico intensivo en números funcionaría bien al mismo tiempo que una larga y intensiva consulta de base de datos.

Por otro lado, si dos procesos utilizan los recursos de manera similar, puede ser menos eficiente ejecutarlos al mismo tiempo que ejecutarlos secuencialmente. ¿Por qué? Básicamente, porque en tales circunstancias, el sistema operativo a menudo tiene que «dividir el tiempo» de los recursos en disputa.

Por ejemplo, si ambos procesos consumen muchos recursos del disco, el sistema operativo puede entrar en un modo en el que controla constantemente el interruptor del disco entre los dos procesos competidores; el sistema termina gastando al menos tanto tiempo haciendo el cambio como en los propios procesos. Este fenómeno se conoce como *thrashing*; en su forma más severa, puede hacer que un sistema prácticamente se paralice. El thrashing es un problema común; tanto los administradores del sistema como los diseñadores de sistemas operativos pasan mucho tiempo tratando de minimizarlo.

8.5.3. Paralelización

Pero si tienes una computadora con múltiples CPUs [11], deberías preocuparte menos por el thrashing. Además, las corrutinas pueden proporcionar aumentos dramáticos de velocidad en este tipo de máquina, que a menudo se llama una computadora *paralela*; de manera análoga, descomponer un proceso en corrutinas a veces se denomina *paralelizar* el trabajo.

Normalmente, cuando inicias un trabajo en segundo plano en una máquina con múltiples CPUs, la computadora lo asigna al siguiente procesador disponible. Esto significa que los dos trabajos están realmente, no solo metafóricamente, ejecutándose al mismo tiempo.

En este caso, el tiempo de ejecución de las corrutinas es esencialmente igual al del trabajo de mayor duración más un poco de sobrecarga, en lugar de la suma de los tiempos de ejecución de todos los procesos (aunque si todas las CPUs comparten una unidad de disco común, aún existe la posibilidad de thrashing relacionado con la E/S). En el mejor caso, con todos los trabajos teniendo el mismo tiempo de ejecución y sin contención de E/S, obtienes un factor de aceleración igual al número de CPUs.

Paralelizar un programa a menudo no es fácil; hay varios problemas sutiles involucrados y hay mucho margen para el error. Sin embargo, vale la pena saber cómo paralelizar un script de shell, ya sea que tengas o no una máquina paralela, especialmente porque estas máquinas son cada vez más comunes, incluso en el escritorio.

[11] Los sistemas multiprocesador solían encontrarse solo en servidores a gran escala ubicados en salas de máquinas especiales con control climático. Hoy en día, los sistemas multiprocesador de escritorio están disponibles y se están volviendo cada vez más comunes, aunque los sistemas con más de alrededor de 4 CPUs aún tienden a estar principalmente en salas de máquinas.

Mostraremos cómo hacer esto mediante la Tarea 8-3, una tarea sencilla cuya solución es propicia para la paralelización.

Tarea 8-3

Aumenta el script del frente del compilador de C para compilar cada archivo fuente en paralelo.

Si tienes múltiples CPUs, hay un potencial considerable para acelerar considerablemente la compilación de múltiples archivos fuente en paralelo. Cada archivo es independiente del siguiente, y por lo tanto, crear múltiples archivos objeto simultáneamente realiza más trabajo, más rápido.

Los cambios son relativamente sencillos: inicia la tubería de compilación en segundo plano y luego agrega una declaración *wait* antes de realizar el paso final de enlace:

```
# initialize option-related variables
do_link=true
debug=""
link_libs=""
clib="-lc"
exefile=""

# initialize pipeline components
compile=" | ccom"
assemble=" | as"
optimize=""

# process command-line options
while getopts "cgl:[lib]o:[outfile]O files ..." opt; do
  case $opt in
    c )
      do_link=false ;;
    g )
      debug="-g" ;;
    l )
      link_libs+=" -l $OPTARG" ;;
    o )
      exefile="-o $OPTARG" ;;
    O )
      optimize=" | optimize" ;;
  esac
done

shift $(($OPTIND - 1))

# process the input files
for filename in "$@"; do
  case $filename in
```

```
    *.c )
      objname=${filename%.c}.o ;;
    *.s )
      objname=${filename%.s}.o compile="" ;;
    *.o )
      objname=$filename # just link it directly with the rest
      compile="" assemble="" ;;
    * )
      print "error: $filename is not a source or object file."
      exit 1 ;;
  esac

  # run a pipeline for each input file; parallelize by backgrounding
  eval cat \$filename $compile $assemble $optimize \> \$objname &
  objfiles+=" $objname"
  compile=" | ccom" assemble=" | as"
done

wait # wait for all compiles to finish before linking
if [[ $do_link == true ]]; then
  ld $exefile $objfiles $link_libs $clib
fi
```

Este es un ejemplo directo de paralelización, siendo la única «trampa» asegurarse de que todas las compilaciones se realicen antes de realizar el paso final de enlace. De hecho, muchas versiones de *make* tienen una bandera de «ejecutar tantos trabajos en paralelo» precisamente para obtener la aceleración de la compilación simultánea de archivos independientes.

Pero no toda la vida es tan simple; a veces, simplemente iniciar más trabajos en segundo plano no hará el truco. Por ejemplo, considera múltiples cambios en la misma base de datos: el software de la base de datos (o algo, en algún lugar) tiene que asegurarse de que dos procesos diferentes no estén intentando actualizar el mismo registro al mismo tiempo.

Las cosas se complican aún más al trabajar a un nivel más bajo, con múltiples hilos de control dentro de un solo proceso (algo que afortunadamente no es visible a nivel de shell). Dichos problemas, conocidos como problemas de *control de concurrencia*, se vuelven mucho más difíciles a medida que aumenta la complejidad de la aplicación. Los programas concurrentes complejos a menudo tienen mucho más código para manejar los casos especiales que para el trabajo real que se supone deben hacer.

Por lo tanto, no debería sorprenderte que se haya hecho y se esté haciendo mucha investigación sobre la paralelización, siendo el objetivo final idear una herramienta que paralelice el código automáticamente. (Tales herramientas existen; generalmente trabajan dentro de

algún subconjunto estrecho del problema). Incluso si no tienes acceso a una máquina con múltiples CPUs, paralelizar un script de shell es un ejercicio interesante que debería familiarizarte con algunos de los problemas que rodean a las corrutinas.

8.5.4. Corrutinas con tuberías bidireccionales

Ahora que hemos visto cómo programar corrutinas que no se comunican entre sí, construiremos sobre esa base y discutiremos cómo hacer que se comuniquen, de una manera más sofisticada que con una canalización. El shell Korn tiene un conjunto de características que permiten a los programadores establecer una comunicación bidireccional entre corrutinas. Estas características no están incluidas en la mayoría de los shells Bourne.

Si inicias un proceso en segundo plano agregando |& a un comando en lugar de &, el shell Korn configura una tubería bidireccional especial entre el shell principal y el nuevo proceso en segundo plano. `read -p` en el shell principal lee una línea de la salida estándar del proceso en segundo plano; de manera similar, `print -p` en el shell principal alimenta la entrada estándar del proceso en segundo plano. La Figura 8.2 muestra cómo funciona esto.

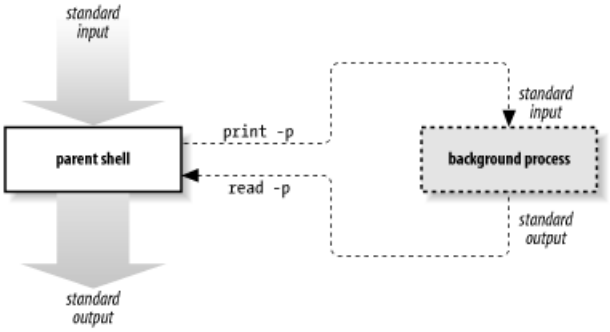

Figura 8.2: E/S de las corrutinas

Este esquema tiene algunas posibilidades intrigantes. Observa lo siguiente: primero, el shell principal se comunica con el proceso en segundo plano de manera independiente de su propia entrada y salida estándar. En segundo lugar, el proceso en segundo plano no necesita tener idea de que un script de shell se está comunicando con él de esta manera. Esto significa que el proceso en segundo plano puede ser casi cualquier programa preexistente que use su entrada y salida estándar de manera normal.[12]

La Tarea 8-4 es una tarea que muestra un ejemplo sencillo.

[12]Observa que *sort(1)* no encaja del todo aquí. *sort* debe leer toda su entrada antes de poder generar cualquier salida. Todavía puedes usar *sort* en un coproceso, pero tendrías que cerrar el descriptor de archivo utilizado para escribir en el coproceso primero. La forma de hacer esto es mover el descriptor de archivo de entrada del coproceso a un descriptor de archivo numerado y luego cerrarlo. Ambas cosas involucran el comando *exec*, que se trata en el próximo capítulo.

Tarea 8-4

Quieres tener una calculadora en línea, pero la utilidad Unix existente *dc(1)* utiliza la Notación Polaca Inversa (RPN), al estilo de las calculadoras Hewlett-Packard. Preferirías tener una que funcione como el modelo de $3.95 que recibiste con esa suscripción a una revista. Escribe un programa de calculadora que acepte notación algebraica estándar.

El objetivo aquí es escribir el programa sin volver a implementar el motor de cálculo que ya tiene *dc*, en otras palabras, escribir un programa que traduzca la notación algebraica a RPN y pase la línea traducida a *dc* para hacer el cálculo real. (La utilidad *bc(1)* proporciona funcionalidad similar).

Supondremos que la función *alg2rpn*, que realiza la traducción, ya existe: dada una línea de notación algebraica como argumento, imprime el equivalente en RPN en la salida estándar. Si tenemos esto, entonces el programa de la calculadora (que llamaremos *adc*) es muy simple:

```
dc |&

while read line'?adc> '; do
  print -p "$(alg2rpn $line)"
  read -p answer
  print " = $answer"
done
```

La primera línea de este código inicia *dc* como una corrutina con una tubería bidireccional. Luego, el bucle `while` solicita al usuario una línea y la lee hasta que el usuario escribe CTRL-D para el final de la entrada. El cuerpo del bucle convierte la línea a RPN, la pasa a *dc* a través de la tubería, lee la respuesta de *dc* e imprime después un signo igual. Por ejemplo:

```
$ adc
adc> 2 + 3
  = 5
adc> (7 * 8) + 54
  = 110
adc> ^D
$
```

En realidad, como habrás notado, no es del todo necesario tener una tubería bidireccional con *dc*. Podrías hacerlo con una tubería estándar y dejar que *dc* haga su propia salida, así:

```
{ while read line'?adc> '; do
  print "$(alg2rpn $line)"
```

```
  done
} | dc
```

La única diferencia con lo anterior es la falta del signo igual antes de imprimir cada respuesta.

¿Pero qué pasa si quisieras hacer una interfaz gráfica de usuario elegante (GUI), como el programa *xcalc* que viene con muchas instalaciones del sistema X Window? Entonces, claramente, la salida propia de *dc* no sería satisfactoria y necesitarías un control total de tu propia salida estándar en el proceso principal. La interfaz de usuario tendría que capturar la salida de *dc* y mostrarla en la ventana de la calculadora de manera adecuada. La tubería bidireccional es una excelente solución a este problema: simplemente imagina que, en lugar de `print "  = $answer"`, hay una llamada a una rutina que muestra la respuesta en la sección de «lectura» de la ventana de la calculadora.

Todo esto sugiere que el esquema de tubería bidireccional es excelente para escribir scripts de shell que interponen una capa de software entre el usuario (o algún otro programa) y un programa existente que utiliza entrada y salida estándar. En particular, es excelente para escribir nuevas interfaces para programas Unix estándar antiguos que esperan entrada y salida de usuario basada en caracteres, línea por línea. Las nuevas interfaces podrían ser GUI o podrían ser programas de interfaz de red que se comuniquen con usuarios a través de enlaces a máquinas remotas. En otras palabras, ¡la construcción de tuberías bidireccionales del shell Korn está diseñada para ayudar a desarrollar software muy actualizado!

8.5.5. Tuberías Bidireccionales versus Tuberías Estándar

Antes de abandonar el tema de las corrutinas, completaremos el círculo mostrando cómo se compara la construcción de tuberías bidireccionales con las canalizaciones regulares. Como probablemente hayas podido deducir hasta ahora, es posible programar una canalización estándar usando |& con `print -p`.

Esto tiene la ventaja de reservar la salida estándar del shell principal para otro uso. La desventaja es que la salida estándar del proceso secundario se dirige a la tubería bidireccional: si el proceso principal no la lee con `read -p`, se pierde efectivamente.

8.6. Subprocesos Shell y Subshell

Las corrutinas representan claramente la relación más compleja entre procesos que define el shell Korn. Para concluir este capítulo, veremos un tipo de relación entre procesos mucho más simple: la de un subproceso de shell con su shell padre. Vimos en el Capítulo 3 que cada vez que ejecuta un script de shell, en realidad invoca otra copia del shell que es un subproceso del proceso de shell principal o principal . Ahora veámoslos con más detalle.

8.6.1. Herencia de subprocesos de shell

Las cosas más importantes que debe saber sobre los subprocesos de shell son las características que obtienen o heredan de sus padres. Estos son los siguientes:

- El directorio actual
- Variables de entorno
- Entrada, salida y error estándar más cualquier otro descriptor de archivo abierto
- Cualquier característica definida en el archivo de entorno (ver Capítulo 3). Tenga en cuenta que solo los shells interactivos ejecutan el archivo de entorno
- Señales que se ignoran

Las tres primeras características son heredadas por todos los subprocesos, mientras que las dos últimas son exclusivas de los subprocesos de shell. Igual de importantes son las cosas que un subproceso de shell no hereda de su padre:

- Variables de shell, excepto las variables de entorno y las definidas en el archivo de entorno
- Manejo de señales que no se ignoran

Cubrimos algo de esto anteriormente (en el Capítulo 3), pero estos puntos son fuentes comunes de confusión, por lo que vale la pena repetirlos.

8.6.2. Subshells

Un tipo especial de subproceso del shell es el subshell. El subshell se inicia dentro del mismo script (o función) que el padre. Lo haces de manera muy similar a los bloques de código que vimos en el último capítulo. Simplemente rodea un código de shell con paréntesis (en lugar de llaves), y ese código se ejecuta en un subshell.

Por ejemplo, aquí está el programa de la calculadora, de arriba, con un subshell en lugar de un bloque de código:

```
( while read line'?adc> '; do
print "$(alg2rpn $line)"
done
) | dc
```

El código dentro de los paréntesis se ejecuta como un proceso separado.[13] Esto suele ser menos eficiente que un bloque de código. Las diferencias en funcionalidad entre subshell y bloques de código son muy pocas; se refieren principalmente a problemas de ámbito, es decir, a los dominios en los que se conocen las definiciones de cosas como variables de shell y trampas de señales. En primer lugar, el código dentro de un subshell obedece a las reglas de la herencia del subproceso del shell mencionadas anteriormente, excepto que conoce las variables definidas en el shell circundante; en cambio, piensa en los bloques como unidades de código que heredan *todo* del shell exterior. En segundo lugar, las variables y trampas definidas dentro de un bloque de código son conocidas por el código de shell después del bloque, mientras que las definidas en un subshell no lo son.

Por ejemplo, considera este código:

```
{
  fred=bob
  trap 'print "You hit CTRL-C!"' INT
}
while true; do
  print "\$fred is $fred"
  sleep 60
done
```

Si ejecutas este código, verás el mensaje `$fred is bob` cada 60 segundos, y si escribes CTRL-C, verás el mensaje, `You hit CTRL-C!`. Deberás escribir CTRL- para detenerlo (no olvides eliminar el archivo core). Ahora cambiémoslo a un subshell:

```
(
  fred=bob
  trap 'print "You hit CTRL-C!"' INT
)
while true; do
  print "\$fred is $fred"
  sleep 60
done
```

[13] Por razones de rendimiento, el shell Korn hace todo lo posible para evitar crear realmente un proceso separado para ejecutar el código entre paréntesis y dentro de `$(...)`. Pero los resultados siempre deben ser los mismos que *si* el código se ejecutara en un proceso separado.

Si ejecutas esto, verás el mensaje `$fred is`; el shell exterior no conoce la definición de *fred* del subshell y, por lo tanto, piensa que es nulo. Además, el shell exterior no conoce la trampa de la señal INT del subshell, así que si presionas CTRL-C, el script se termina.

Si un lenguaje admite el anidamiento de código, las definiciones dentro de una unidad anidada deberían tener un ámbito limitado a esa unidad anidada. En otras palabras, los subshell te brindan un mejor control que los bloques de código sobre el ámbito de las variables y las trampas de señales. Por lo tanto, creemos que debes usar subshell en lugar de bloques de código si van a contener definiciones de variables o trampas de señales, a menos que la eficiencia sea una preocupación.

Este ha sido un capítulo largo y ha cubierto mucho terreno. Aquí tienes algunos ejercicios que deberían ayudarte a asegurarte de que tienes un firme entendimiento del material. El último ejercicio es especialmente difícil para aquellos sin antecedentes en compiladores, teoría del análisis sintáctico o teoría formal de lenguajes.

1. Escribe una función llamada *pinfo* que combine los comandos *jobs* y *ps* imprimiendo una lista de trabajos con sus números de trabajo, identificadores de proceso correspondientes, tiempos de ejecución y comandos completos. Puntuación extra: describe por qué esto debe ser una función y no un script.

2. Toma la última versión de nuestro script de shell del compilador C, o algún otro script de shell no trivial, y «a prueba de balas» con trampas de señales.

3. Vuelve a hacer el programa *findterms* en el último capítulo usando un subshell en lugar de un bloque de código.

4. Lo siguiente no tiene mucho que ver con el material de este capítulo en sí, pero es un ejercicio de programación clásico y te dará práctica si lo haces:

   a) Escribe la función *alg2rpn* utilizada en *adc*. Así es como debes hacerlo: las expresiones aritméticas en notación algebraica tienen la forma *expr op expr*, donde cada *expr* es o bien un número o bien otra expresión (quizás entre paréntesis), y *op* es +, -, x, / o % (resto). En RPN, las expresiones tienen la forma *expr expr op*. Por ejemplo: la expresión algebraica $2 + 3$ es 23+ en RPN; el equivalente en RPN de $(2 + 3)x(9 - 5)$ es $23 + 95 - x$. La principal ventaja de RPN es que elimina la necesidad de paréntesis y reglas de precedencia de operadores (por ejemplo, la regla de que x se evalúa antes que +). El programa *dc* acepta RPN estándar, pero cada expresión debe tener «p» al final: esto le dice a *dc* que

imprima su resultado, por ejemplo, el primer ejemplo anterior debería darse a *dc* como $23 + p$.

b) Necesitas escribir una rutina que convierta la notación algebraica en RPN. Esto debería ser (o incluir) una función que se llame a sí misma (conocida como una función *recursiva*) cada vez que encuentra una subexpresión. Es especialmente importante que esta función lleve un registro de dónde está en la cadena de entrada y cuánto de la cadena consume durante su procesamiento. (Pista: utiliza los operadores de coincidencia de patrones discutidos en el Capítulo 4 para facilitar la tarea de analizar cadenas de entrada).

Para facilitarte la vida, no te preocupes por la precedencia de operadores por ahora; simplemente convierte a RPN de izquierda a derecha. Por ejemplo, trata $3+4x5$ como $(3+4)x5$ y $3x4+5$ como $(3x4)+5$. Esto hace posible que conviertas la cadena de entrada sobre la marcha, es decir, sin tener que leerla completa antes de hacer cualquier procesamiento.

c) Mejora tu solución al ejercicio anterior para que admita la precedencia de operadores en el orden usual: x, /, % (resto) +, -. Por ejemplo, trata $3 + 4x5$ como $3 + (4x5)$ y $3x4 + 5$ como $(3x4) + 5$.

CAPÍTULO 9

DEPURACIÓN DE PROGRAMAS DEL SHELL

Esperamos haber logrado convencerte de que el shell Korn puede utilizarse como un entorno de programación Unix serio. Ciertamente, tiene muchas características, estructuras de control, etc. Pero otra parte esencial de un entorno de programación son las herramientas de *soporte integradas y potentes*. Por ejemplo, hay una amplia variedad de editores de pantalla, compiladores, depuradores, perfiles, referenciadores cruzados, etc., para lenguajes como C, C++ y Java. Si programas en alguno de estos lenguajes, probablemente das por sentadas tales herramientas, y sin duda te horrorizarías ante la idea de tener que desarrollar código con, por ejemplo, el editor *ed* y el depurador de lenguaje máquina *adb*.

Pero, ¿qué hay de las herramientas de soporte para el Korn shell? Por supuesto, puedes usar cualquier editor que desees, incluyendo *vi* y *Emacs*. Y debido a que el shell es un lenguaje interpretado, no necesitas un compilador.[1] Pero no hay otras herramientas disponibles. El problema más serio es la falta de un depurador.

Este capítulo aborda esa carencia. El shell tiene algunas características que ayudan en la depuración de scripts de shell; veremos estas en la primera parte del capítulo. El shell Korn también tiene un par de características nuevas, no presentes en la mayoría de los shells de Bourne, que hacen posible implementar una herramienta de depuración completa. Mostramos estas características; más importante aún, presentamos *kshdb*, un depurador de Korn shell que las utiliza. *kshdb* es básico pero bastante utilizable, y su implementación sirve como un ejemplo extendido de varias técnicas de programación de shell de todo este libro.

[1] De hecho, si realmente te preocupa la eficiencia, hay compiladores de código de shell en el mercado; algunos convierten scripts de shell a código C que a menudo se ejecuta bastante más rápido; sin embargo, estas herramientas suelen ser para scripts de Bourne shell. Otros «compiladores» simplemente convierten el script a una forma binaria para que los clientes no puedan leer el programa.

9.1. Ayudas básicas para depuración

¿Qué tipo de funcionalidad necesitas para depurar un programa? En el nivel más empírico, necesitas una forma de determinar *qué* está causando que tu programa se comporte mal y dónde está el problema en el código. Por lo general, comienzas con un *qué* obvio (como un mensaje de error, una salida inapropiada, un bucle infinito, etc.), intentas retroceder hasta encontrar un ”qué”que esté más cerca del problema real (por ejemplo, una variable con un valor incorrecto, una opción incorrecta para un comando) y eventualmente llegas al ”dónde.[ex]acto en tu programa. Luego puedes preocuparte por *cómo* solucionarlo.

Observa que estos pasos representan un proceso de empezar con información obvia y terminar con hechos a menudo oscuros deducidos e intuidos. Las ayudas para la depuración facilitan la deducción e intuición al proporcionar información relevante fácilmente o incluso automáticamente, preferiblemente sin modificar tu código.

La ayuda más simple para la depuración (para cualquier lenguaje) es la instrucción de salida, como *print* en el caso del shell. De hecho, los programadores de la vieja escuela depuraban su código Fortran insertando tarjetas WRITE en sus mazos. Puedes depurar colocando muchas declaraciones de salida en tu código (y eliminándolas más tarde), pero tendrás que dedicar mucho tiempo a reducir no solo la información exacta que deseas sino también dónde necesitas verla. También probablemente tendrás que sumergirte en mucha salida para encontrar la información que realmente deseas.

9.1.1. Establecer opciones

Afortunadamente, el shell tiene algunas características básicas que te brindan funcionalidad de depuración más allá de la de *print*. Las más básicas son las opciones del comando `set -o` (como se cubrió en el Capítulo 3). Estas opciones también se pueden usar en la línea de comandos al ejecutar un script, como muestra la Tabla 9.1.

La opción *verbose* simplemente imprime (en el error estándar) cualquier entrada que reciba el shell. Es útil para encontrar el punto exacto en el que un script está fallando. Por ejemplo, supongamos que tu script se ve así:

```
fred
bob
dave
pete
ed
ralph
```

Tabla 9.1: Opciones de depuración

Opción `set -o`	**Opción de línea de comandos**	**Acción**
noexec	-n	No ejecute comandos; compruebe sólo errores de sintaxis
verbose	-v	Eco de comandos antes de ejecutarlos
xtrace	-x	Comandos de eco tras el procesamiento de la línea de comandos

Ninguno de estos comandos son programas Unix estándar, y todos realizan su trabajo en silencio. Digamos que el script falla con un mensaje críptico como «violación de segmento». Esto no te dice nada sobre qué comando causó el error. Si escribes `ksh -v nombredelscript`, podrías ver esto:

```
fred
bob
dave
segmentation violation
pete
ed
ralph
```

Ahora sabes que *dave* es el probable culpable, aunque también es posible que *dave* haya fallado debido a algo que esperaba que *fred* o *bob* hicieran (por ejemplo, crear un archivo de entrada) que hicieron incorrectamente.

La opción *xtrace* es más potente: imprime cada comando y sus argumentos después de que el comando ha pasado por la sustitución de parámetros, la sustitución de comandos y los demás pasos del procesamiento de la línea de comandos (como se describe en el Capítulo 7). Si es necesario, la salida se cita de tal manera que se pueda reutilizar más tarde como entrada para el shell.

Aquí tienes un ejemplo:

```
$ set -o xtrace
$ fred=bob
+ fred=bob
$ print "$fred"
+ print bob
bob
$ ls -l $(whence emacs)
+ whence emacs
+ ls -l /usr/bin/emacs
-rwxr-xr-x 2    root  root      3471896 Mar 16 20:17 /usr/bin/emacs
$
```

Como puedes ver, *xtrace* inicia cada línea que imprime con +. Esto es realmente personalizable: es el valor de la variable de shell integrada PS4.[2] Si estableces PS4 en `xtrace->` (por ejemplo, en tu *.profile* o archivo de entorno), obtendrás listados de *xtrace* que se ven así:

```
$ ls -l $(whence emacs)
xtrace-> whence emacs
xtrace-> ls -l /usr/bin/emacs
-rwxr-xr-x    2  root    root    3471896 Mar 16 20:17  /usr/bin/emacs
$
```

Una forma aún mejor de personalizar PS4 es usar una variable integrada que aún no hayamos visto: `LINENO`, que contiene el número de la línea que se está ejecutando actualmente en un script de shell. Coloca esta línea en tu *.profile* o archivo de entorno:

```
PS4='line $LINENO: '
```

Usamos la misma técnica que con PS1 en el Capítulo 3: usando comillas simples para posponer la evaluación de la cadena hasta cada vez que el shell imprime el indicador. Esto imprime mensajes en la forma `line N:` en tu salida de *trace*. Incluso podrías incluir el nombre del script de shell que estás depurando en este indicador usando el parámetro posicional $0:

```
PS4='$0 line $LINENO: '
```

Como otro ejemplo, digamos que estás tratando de rastrear un error en un script llamado *fred* que contiene este código:

```
dbfmq=$1.fmq
...
fndrs=$(cut -f3 -d' ' $dfbmq)
```

Escribes *fred bob* para ejecutarlo de la manera normal, y se cuelga. Luego escribes ‘ textttksh -x fred bob, y ves esto:

```
+ dbfmq=bob.fmq
...
+ + cut -f3 -d
```

Se cuelga nuevamente en este punto. Notas que *cut* no tiene un argumento de nombre de archivo, lo que significa que debe haber algo mal con la variable *dbfmq*. Pero ha ejecutado correctamente la instrucción de asignación `dbfmq=bob.fmq`... ¡ah, ja! Cometiste un

[2] Como con PS1 y PS3, esta variable también se somete a la sustitución de parámetros, comandos y aritmética antes de que se imprima su valor.

error tipográfico en el nombre de la variable dentro de la construcción de sustitución de comandos.[3] Lo corriges y el script funciona correctamente.

Cuando se establece a nivel global, la opción *xtrace* se aplica al script principal y a cualquier función de estilo POSIX (aquellas creadas con la sintaxis de `nombre ()`). Si el código que estás tratando de depurar llama a funciones de estilo `function` que están definidas en otro lugar (por ejemplo, en tu *.profile* o archivo de entorno), puedes rastrear estas de la misma manera con una opción al comando *typeset*. Simplemente escribe el comando `typeset -ft nombrefuncion` y la función con el nombre dado se rastreará cada vez que se ejecute. Escribe `typeset +ft nombrefuncion` para apagar el rastreo. También puedes poner `set -o xtrace` en el cuerpo de la función, lo cual es bueno cuando la función está dentro del script que se está depurando.

La última opción es *noexec*, que lee el script de shell y verifica errores de sintaxis pero no ejecuta nada. Vale la pena usarla si tu script es sintácticamente complejo (muchos bucles, bloques de código, operadores de cadenas, etc.) y el error tiene efectos secundarios (como crear un archivo grande o colgar el sistema).

Puedes activar estas opciones con `set -o` en tus scripts de shell, y, como se explica en el Capítulo 3, desactivarlas con `set +o opción`. Por ejemplo, si estás depurando un script con un efecto secundario desagradable, y lo has localizado en un cierto fragmento de código, puedes preceder ese fragmento con `set -o xtrace` (y, tal vez, cerrarlo con `set +o xtrace`) para observarlo con más detalle.

NOTA: La opción *noexec* es una opción «unidireccional». ¡Una vez activada, no puedes desactivarla! Esto se debe a que el shell solo imprime comandos y no los ejecuta. Esto incluye el comando `set +o noexec` que querrías usar para desactivar la opción. Afortunadamente, esto solo se aplica a scripts de shell; el shell ignora esta opción cuando es interactivo.

9.1.2. Señales falsas

Un conjunto más sofisticado de ayudas para depuración son las «señales de depuración simuladas en el shel» que pueden usarse en declaraciones *trap* para que el shell actúe bajo ciertas condiciones. Recuerda del capítulo anterior que *trap* te permite instalar algún código que se ejecuta cuando se envía una señal específica a tu script.

Las señales simuladas actúan como señales reales, pero son generadas por el shell (a di-

[3] Deberíamos admitir que si hubieras activado la opción *nounset* en la parte superior de este script, el shell habría señalado este error.

ferencia de las señales reales, que el sistema operativo subyacente genera). Representan eventos en tiempo de ejecución que probablemente sean interesantes para depuradores, ya sean humanos o herramientas de software, y pueden tratarse de la misma manera que las señales reales dentro de scripts de shell. Se enumeran en la Tabla 9.2.

Tabla 9.2: Señales simuladas

Señal simulada	**Cuando se envía**
EXIT	El shell sale de una función o script
ERR	Un comando devuelve un estado de salida distinto de cero
DEBUG	Antes de cada declaración (después en *ksh88*)
KEYBD	Al leer caracteres en los modos de edición (no para depuración)

La señal KEYBD no se usa para depuración. Es una función avanzada, para la cual posponemos la discusión hasta el Capítulo 10.

EXIT

La trampa EXIT, cuando se establece, ejecuta su código cuando la función o script en el que se estableció sale. Aquí hay un ejemplo simple:

```
function func {
  trap 'print "saliendo de la función"' EXIT
  print 'inicio de la función'
}

trap 'print "saliendo del script"' EXIT
print 'inicio del script'
func
```

Si ejecutas este script, verás esta salida:

```
inicio del script
inicio de la función
saliendo de la función
saliendo del script
```

En otras palabras, el script comienza estableciendo la trampa para su propia salida. Luego imprime un mensaje y finalmente llama a la función. La función hace lo mismo, establece una trampa para su salida e imprime un mensaje. (Recuerda que las funciones de estilo de `function` pueden tener sus propias trampas locales que anulan cualquier trampa establecida por el script circundante, mientras que las funciones POSIX comparten trampas con el script principal).

La función luego sale, lo que hace que el shell le envíe la señal simulada EXIT, que a su vez ejecuta el código `print "saliendo de la función"`. Luego, el script sale, y se ejecuta

su propio código de trampa EXIT. Observa también que las trampas «se apilan»; la señal simulada EXIT se envía a cada función en ejecución a medida que cada función llamada más recientemente sale.

Una trampa EXIT ocurre sin importar cómo salga el script o la función, ya sea de manera normal (al finalizar la última instrucción), mediante una declaración explícita de *salida* o *retorno*, o al recibir una señal «real» como INT o TERM. Considera el siguiente programa absurdo de adivinanzas de números:

```
trap 'print "¡Gracias por jugar!"' EXIT

magicnum=$(($RANDOM%10+1))
print 'Adivina un número entre 1 y 10:'
while read guess'?número> '; do
  sleep 10
  if (( $guess == $magicnum )); then
    print '¡Correcto!'
    exit
  fi
  print 'Incorrecto.'
done
```

Este programa elige un número entre 1 y 10 obteniendo un número aleatorio (a través de la variable integrada `RANDOM`, consulta el Apéndice B), extrayendo el último dígito (el resto al dividir por 10) y sumando 1. Luego te pide una adivinanza y, después de 10 segundos, te dice si adivinaste correctamente.

Si lo haces, el programa sale con el mensaje «¡Gracias por jugar!», es decir, ejecuta el código de la trampa EXIT. Si te equivocas, te pide de nuevo y repite el proceso hasta que aciertes. Si te aburres de este pequeño juego y presionas CTRL-C mientras esperas a que te diga si acertaste, también verás el mensaje.

ERR

La señal simulada `ERR` te permite ejecutar código cada vez que un comando en el script o función circundante sale con un estado distinto de cero. El código de la trampa ERR puede aprovechar la variable integrada ?, que contiene el estado de salida del comando anterior. Sobrevive a la trampa y es accesible al principio del código de manejo de la trampa.

Un uso simple pero efectivo de esto es colocar el siguiente código en un script que desees depurar:

```
function errtrap {
  typeset es=$?
```

```
  print "ERROR: El comando salió con estado $es."
}

trap errtrap ERR
```

La primera línea guarda el estado de salida distinto de cero en la variable local `es`.

Por ejemplo, si el shell no puede encontrar un comando, devuelve un estado 1. Si colocas el código en un script con una línea de sin sentido (como «lskdjfafd»), el shell responde con:

```
scriptname: line N: lskdjfafd: not found
ERROR: command exited with status 1.
```

N es el número de la línea en el script que contiene el comando incorrecto. En este caso, el shell imprime el número de línea como parte de su propio mecanismo de informe de errores, ya que el error fue un comando que el shell no pudo encontrar. Pero si el estado de salida distinto de cero proviene de otro programa, el shell no informa el número de línea. Por ejemplo:

```
function errtrap {
  typeset es=$?
  print "ERROR: El comando salió con estado $es."
}

trap errtrap ERR

function bad {
  return 17
}

bad
```

Esto solo imprime `ERROR: El comando salió con estado 17.`

Sería obviamente una mejora incluir el número de línea en este mensaje de error. La variable integrada `LINENO` existe, pero si la usas dentro de una función, se evalúa como el número de línea en la función, no en el archivo en general. En otras palabras, si usas `$LINENO` en la declaración *print* en el procedimiento *errtrap*, siempre se evaluará como 2.

Para resolver este problema, simplemente pasamos `$LINENO` como argumento al controlador de trampas, rodeándolo con comillas simples para que no se evalúe hasta que la señal simulada realmente llegue:

```
function errtrap {
  typeset es=$?
  print "ERROR línea $1: El comando salió con estado $es."
}
trap 'errtrap $LINENO' ERR
```

Si usas esto con el ejemplo anterior, el resultado es el mensaje `ERROR línea 12: El comando salió con estado 17`. Esto es mucho más útil. Veremos una variación de esta técnica en breve.

Este código simple no es realmente un mal mecanismo de depuración universal. Toma en cuenta que un estado de salida distinto de cero no necesariamente indica una condición o evento indeseable: recuerda que cada construcción de control con una condición (`if`, `while`, etc.) usa un estado de salida distinto de cero para significar «falso». En consecuencia, el shell no genera trampas *ERR* cuando las declaraciones o expresiones en las partes «condición» de las estructuras de control producen estados de salida distintos de cero.

Pero una desventaja es que los estados de salida no son tan uniformes (o incluso tan significativos) como deberían ser, como explicamos en el Capítulo 5. Un estado de salida específico no tiene por qué decir nada sobre la naturaleza del error o incluso que hubo un error.

DEBUG

La última señal simulada relacionada con la depuración, DEBUG, provoca que el código de la trampa se ejecute antes de cada declaración en la función o script circundante.[4] Esto tiene dos posibles usos. El primero es para los humanos, como una especie de método «brutal» para rastrear un cierto elemento del estado de un programa que notas que está yendo mal.

Por ejemplo, notas que el valor de una variable en particular está fuera de control. El enfoque ingenuo sería insertar muchas declaraciones de *print* para verificar el valor de la variable en varios puntos. La trampa DEBUG facilita esto:

```
function dbgtrap {
  print "badvar es $badvar"
}

trap dbgtrap DEBUG

... Sección de código donde ocurre el problema ...
trap - DEBUG            # desactivar la trampa DEBUG
```

Este código imprime el valor de la variable problemática antes de cada declaración entre las dos *trampas*.

El segundo y mucho más importante uso de la trampa DEBUG es como un primitivo para

[4] Esto es un cambio notable desde *ksh88*, donde la trampa se ejecutaba *después* de cada declaración.

implementar depuradores de Korn shell. De hecho, sería justo decir que la trampa DEBUG reduce la tarea de implementar un depurador de shell útil de un proyecto de desarrollo de software a una tarea manejable. Hablaremos de esto en breve.

Orden de entrega de señales

Es posible que múltiples señales lleguen simultáneamente (o cerca). En ese caso, el shell ejecuta los comandos de trampa en el siguiente orden:

1. DEBUG
2. ERR
3. Señales Unix reales, en orden de número de señal
4. EXIT

9.1.3. Funciones de disciplina

En el Capítulo 4, presentamos la notación de variables compuestas del shell Korn, como `${person.nanme}`. Utilizando esta notación, *ksh93* proporciona funciones especiales, llamadas *funciones de disciplina*, que te brindan control sobre las variables cuando se hacen referencias, se asignan y se desestablecen. Versiones simples de tales funciones podrían verse así:

```
dave=dave                                # Crear la variable
function dave.set {                      # Llamado cuando se asigna a dave
  print "dave acaba de ser asignado como '${.sh.value}'"
}

function dave.get {                      # Llamado cuando se recupera $dave
  print "valor de dave referenciado, es '$dave'"   # esto es seguro

  .sh.value="dave estuvo aquí"           # Cambiar lo que $dave devuelve, dave no cambió
}

function dave.unset {                    # Llamado cuando se desestablece dave
  print "adiós dave!"
  unset dave        # de hecho, haz que dave desaparezca
}
```

NOTA: La función de disciplina *unset* debe usar realmente el comando *unset* para desestablecer la variable; esto no provoca un bucle infinito. De lo contrario, la variable no se desestablecerá, lo que a su vez conduce a un comportamiento muy sorprendente.

Esto es lo que sucede una vez que todas estas funciones están en su lugar:

```
$ print $dave
valor de dave referenciado, es 'dave'                                    # Desde dave.get
dave estuvo aquí                                                         # Desde print
$ dave='¿quién es este tipo dave, de todos modos?'
dave acaba de ser asignado como '¿quién es este tipo dave, de todos modos?' # Desde dave.set
$ unset dave
adiós dave!                                                              # Desde dave.unset
$ print $dave

$
```

Las funciones de disciplina solo se pueden aplicar a variables globales. No se pueden usar con variables locales, aquellas que creas con *typeset* dentro de una función de estilo de función.

La Tabla 9.3 resume las funciones de disciplina integradas.

Tabla 9.3: Funciones de disciplina predefinidas

Nombre	**Propósito**
variable.get	Llamada cuando se recupera el valor de una variable. La asignación a `.sh.value` cambia el valor devuelto pero no la variable en sí.
variable.set	Llamada cuando se asigna una variable. `${.sh.value}` es el nuevo valor que se asigna. Asignar a `.sh.value` cambia el valor que se está asignando.
variable.unset	Llamada cuando una variable se desestablece. Esta función debe usar unset en la variable para que realmente se desactive.

Como acabamos de ver, dentro de las funciones de disciplina, hay dos variables especiales que el shell establece y que te brindan información, así como una variable que puedes establecer para cambiar el comportamiento del shell. La Tabla 9.4 describe estas variables y lo que hacen.

Tabla 9.4: Variables especiales para usar en funciones de disciplina

Variable	**Propósito**
.sh.name	Nombre de la variable para la que se ejecuta la función de disciplina.
.sh.subscript	El subíndice actual de una variable de matriz. (Las funciones de disciplina se aplican a toda la matriz, no a cada elemento con subíndice).
.sh.value	El nuevo valor que se asigna en una función de disciplina set. Si se asigna en una función de disciplina get, cambia el valor devuelto.

A primera vista, no está claro cuál es el valor de las funciones de disciplina. Pero son perfectas para implementar una característica de depuración muy útil, llamada puntos de observación *watchpoints*. Ahora estamos listos para empezar a escribir nuestro depurador de scripts de shell.

9.2. Un depurador para Korn Shell

Los depuradores disponibles comercialmente ofrecen mucha más funcionalidad que las opciones *establecidas* y las señales simuladas del shell. Los más avanzados tienen interfaces gráficas fabulosas, compiladores incrementales, evaluadores simbólicos y otras comodidades por el estilo. Pero prácticamente todos los depuradores modernos, incluso los más modestos, tienen funciones que te permiten «asomarte» a un programa mientras se ejecuta, para examinarlo en detalle y en términos de su lenguaje fuente. Específicamente, la mayoría de los depuradores te permiten hacer estas cosas:

- Especificar puntos en los que el programa detiene la ejecución y entra en el depurador. Estos se llaman puntos de interrupción *breakpoints*.
- Ejecutar solo una parte del programa a la vez, generalmente medida en declaraciones de código fuente. Esta capacidad se llama a menudo paso a paso *stepping*.
- Examinar y posiblemente cambiar el estado del programa (por ejemplo, valores de variables) en medio de una ejecución, es decir, cuando se detiene en un punto de interrupción o después de un paso a paso.
- Especificar variables cuyos valores deben imprimirse cuando se cambian o acceden. Estas se llaman a menudo puntos de observación (*watchpoints*).
- Hacer todo lo anterior sin tener que cambiar el código fuente.

Nuestro depurador, llamado *kshdb*, tiene estas características y algunas más. Aunque es una herramienta básica, sin demasiadas florituras, no es un juguete. El sitio web del libro, http://www.oreilly.com/catalog/korn2/, tiene un enlace para descargar una copia de todos los programas de ejemplo del libro, incluido *kshdb*. Si no tienes acceso a Internet, puedes escribir o escanear el código. De cualquier manera, puedes usar *kshdb* para depurar tus propios scripts de shell, y debes sentirte libre de mejorarlo. Esta es la versión 2.0 del depurador. Incluye algunos cambios sugeridos por Steve Alston, y la función de puntos de observación es completamente nueva. Sugeriremos algunas mejoras al final de este capítulo.

9.2.1. Estructura del depurador

El código para *kshdb* tiene varias características que vale la pena explicar con algún detalle. La más importante es el principio básico en el que funciona: convierte un script de shell en un depurador para sí mismo, agregando funcionalidad de depuración al principio y luego

ejecuta el nuevo script.

El script controlador

Por lo tanto, el código tiene dos partes: la que implementa la funcionalidad del depurador y la que instala esa funcionalidad en el script que se está depurando. La segunda parte, que veremos primero, es el script llamado *kshdb*. Es muy simple:

```
# kshdb -- Depurador de Korn Shell
# Controlador principal: construye el script completo (con encabezado) y lo ejecuta

print "Depurador de Korn Shell versión 2.0 para ksh '${.sh.version}'" >&2
_guineapig=$1
if [[ ! -r $1 ]]; then
  # archivo no encontrado o no legible
  print "No se puede leer $_guineapig." >&2
  exit 1
fi
shift

_tmpdir=/tmp
_libdir=.                    # establecer a un directorio real al instalar
_dbgfile=$_tmpdir/kshdb$$    # archivo temporal para el script que se está depurando (copia)
cat $_libdir/kshdb.pre $_guineapig > $_dbgfile
exec ksh $_dbgfile $_guineapig $_tmpdir $_libdir "$@"
```

kshdb toma como argumento el nombre del script que se está depurando, al que, por brevedad, llamaremos <,programa de pruebas». Cualquier argumento adicional se pasa al programa de pruebas como sus parámetros posicionales. Observa que `${.sh.version}` indica la versión del shell Korn para el mensaje de inicio.

Si el argumento es inválido (el archivo no es legible), *kshdb* sale con un estado de error. De lo contrario, después de un mensaje introductorio, construye un nombre de archivo temporal como vimos en el Capítulo 8. Si no tienes (o no tienes acceso a) `/tmp` en tu sistema, puedes sustituir un directorio diferente por `_tmpdir`. [5] Además, asegúrate de que `_libdir` esté configurado con el directorio donde residen los archivos *kshdb.pre* y *kshdb.fns* (que veremos pronto). `/usr/share/lib` es una buena opción si tienes acceso a él.

La instrucción *cat* construye el archivo temporal: consiste en un archivo que veremos pronto llamado *kshdb.pre*, que contiene el código real del depurador, seguido inmediatamente por una copia del programa en cuestión. Por lo tanto, el archivo temporal contiene un script

[5] Todos los nombres de funciones y variables (excepto los locales de las funciones) en *kshdb* comienzan con un guion bajo (_), para minimizar la posibilidad de conflictos con los nombres en programa de pruebas. Una solución más orientada a *ksh93* sería usar una variable compuesta, por ejemplo, `_db.tmpdir`, `_db.libdir`, y así sucesivamente.

de shell que se ha convertido en un depurador para sí mismo.

exec

La última línea ejecuta este script con *exec*, una instrucción que aún no hemos visto. Hemos elegido esperar hasta ahora para presentarla porque, como creemos que estarás de acuerdo, puede ser peligrosa. *exec* toma sus argumentos como una línea de comandos y ejecuta el comando en lugar del programa actual, en el mismo proceso. En otras palabras, el shell que ejecuta el script anterior *terminará inmediatamente* y será reemplazado por los argumentos de *exec*. Las situaciones en las que querrías usar *exec* son pocas, raras y bastante arcanas, aunque esta es una de ellas.

En este caso, *exec* simplemente ejecuta el script de shell recién construido, es decir, el programa de pruebas con su depurador, en otro shell Korn. Pasa al nuevo script tres argumentos: los nombres del programa de pruebas original (`$_guineapig`), el directorio temporal (`$_tmpdir`), y el directorio donde se mantienen *kshdb.pre* y *kshdb.fns*, seguidos de los parámetros posicionales del usuario, si los hay.

exec también se puede usar solo con un redireccionador de E/S; esto hace que el redireccionador tenga efecto durante el resto del script o sesión de inicio. Por ejemplo, la línea `exec 2>errlog` en la parte superior de un script dirige la propia salida de error estándar del shell al archivo *errlog* durante todo el script. Esto también se puede usar para mover la entrada o salida de un coproceso a un descriptor de archivo numerado normal. Por ejemplo, `exec 5<&p` mueve la salida del coproceso (que es entrada para el shell) al descriptor de archivo 5. De manera similar, `exec 6>&p` mueve la entrada del coproceso (que es salida del shell) al descriptor de archivo 6. El alias predefinido `redirect='command exec'` es más mnemotécnico.

9.2.2. El preámbulo

Ahora veremos el código que se agrega al script que se está depurando; lo llamamos *preámbulo*. Se mantiene en el siguiente archivo, *kshdb.pre*, que también es bastante simple:

```
# preámbulo de kshdb para la versión 2.0 de kshdb
# se antepone al script de shell que se está depurando
# argumentos:
# $1 = nombre del script original de pruebas
# $2 = directorio donde se almacenan los archivos temporales
# $3 = directorio donde se almacenan kshdb.pre y kshdb.fns
_dbgfile=$0
_guineapig=$1
```

```
_tmpdir=$2
_libdir=$3
shift 3                         # mueve los argumentos del usuario al lugar correspondiente
. $_libdir/kshdb.fns            # lee las funciones de depuración
_linebp=
_stringbp=
let _trace=0                    # inicializa el rastreo de ejecución como apagado
typeset -A _lines
let _i=1                        # lee el archivo programa de pruebas en el array de líneas
while read -r _lines[$_i]; do
  let _i=$_i+1
done < $_guineapig
trap _cleanup EXIT              # borra archivos antes de salir
let _steps=1                    # no. de declaraciones para ejecutar después de que se establece
    la trampa
LINENO=0
trap '_steptrap $LINENO' DEBUG
```

Las primeras líneas guardan los tres argumentos fijos en variables y los desplazan para que los parámetros posicionales (si los hay) sean los que el usuario proporcionó en la línea de comandos como argumentos para programa de pruebas. Luego, el preámbulo lee otro archivo, *kshdb.fns*, que contiene la esencia del depurador como definiciones de funciones. Ponemos este código en un archivo separado para minimizar el tamaño del archivo temporal. Examinaremos *kshdb.fns* en breve.

A continuación, *kshdb.pre* inicializa las dos listas de puntos de interrupción como vacías y el rastreo de ejecución como apagado (ver más abajo), luego lee el programa de pruebas en un array de líneas. Hacemos esto último para que el depurador pueda acceder a las líneas en el script al realizar ciertas verificaciones y para que la función de rastreo de ejecución pueda imprimir líneas de código a medida que se ejecutan. Utilizamos un array asociativo para mantener el código fuente del script de shell, para evitar el límite incorporado (aunque grande) de 4096 elementos para arrays indexados. (Admitimos que nuestro uso es un poco inusual; usamos números de línea como índices, pero en lo que respecta al shell, estos son solo cadenas que contienen solo dígitos).

La verdadera diversión comienza en el último grupo de líneas de código, donde configuramos el depurador para que comience a funcionar. Utilizamos dos comandos *trap* con señales falsas. El primero establece una rutina de limpieza (que simplemente borra el archivo temporal) que se llamará en EXIT, es decir, cuando el script termine por cualquier motivo. El segundo, y más importante, configura la función *_steptrap* para que se llame antes de cada declaración.

_steptrap recibe un argumento que se evalúa como el número de la línea en programa de pruebas que se acaba de ejecutar. Usamos la misma técnica con la variable integrada `LINENO` que vimos anteriormente en el capítulo, pero con un giro añadido: si asignas un valor a `LINENO`, lo utiliza como el próximo número de línea e incrementa a partir de ahí. La instrucción `LINENO=0` reinicia la numeración de líneas para que la primera línea en programa de pruebas sea la línea 1.

Después de que se establece la trampa DEBUG, termina el preámbulo. La trampa DEBUG se ejecuta *antes* de la siguiente declaración, que es la primera declaración del programa de pruebas. El shell entra así en *_steptrap* por primera vez. La variable *_steps* se configura para que *_steptrap* ejecute su última cláusula `elif`, como verás en breve, y entre en el depurador. Como resultado, la ejecución se detiene justo antes de que se ejecute la primera declaración del programa de pruebas, y el usuario ve un indicador `kshdb>`; el depurador está ahora en pleno funcionamiento.

9.2.3. Funciones del depurador

La función *_steptrap* es el punto de entrada al depurador; está definida en el archivo *kshdb.fns*, que se enumera por completo al final de este capítulo. Aquí está *_steptrap*:

```
# Aquí antes de cada declaración en el script que se está depurando.
# Maneja paso a paso y puntos de interrupción.
function _steptrap {
  _curline=$1                       # el argumento es el número de la línea que se ejecutó
  (( $_trace )) && _msg "$PS4 línea $_curline: ${_lines[$_curline]}"
  if (( $_steps >= 0 )); then       # si está en modo paso a paso
    let _steps="$_steps - 1"        # decrementa el contador
  fi

  # primera verificación: si se alcanza el punto de interrupción de número de línea
  if _at_linenumbp; then
    _msg "Se alcanzó el punto de interrupción de línea en la línea $_curline"
    _cmdloop                        # punto de interrupción, entra en el depurador

  # segunda verificación: si se alcanza el punto de interrupción de cadena
  elif _at_stringbp; then
    _msg "Se alcanzó el punto de interrupción de cadena en la línea $_curline"
    _cmdloop                        # punto de interrupción, entra en el depurador

  # si no, verifica si existe y es verdadera la condición de interrupción
  elif [[ -n $_brcond ]] && eval $_brcond; then
    _msg "La condición de interrupción '$_brcond' es verdadera en la línea $_curline"
    _cmdloop                        # condición de interrupción, entra en el depurador

  # finalmente, verifica si está en modo paso a paso y el número de pasos ha terminado
```

```
  elif (( _steps == 0 )); then      # si está en modo paso a paso y es el momento de detenerse
    _msg "Detenido en la línea $_curline"
    _cmdloop                        # entra en el depurador
  fi
}
```

_steptrap comienza estableciendo *_curline* en el número de la línea del programa de pruebas que se acaba de ejecutar. Si el rastreo de ejecución está activado, imprime el indicador de rastreo de ejecución PS4 (a la manera del modo *xtrace*), el número de línea y la línea de código en sí.

Luego hace una de dos cosas: entra en el depurador, cuyo corazón es la función *_cmdloop*, o simplemente devuelve el control para que el shell pueda ejecutar la siguiente declaración. Elige lo primero si se ha alcanzado un punto de interrupción o una condición de interrupción (ver más abajo), o si el usuario entró en esta declaración.

Comandos

Explicaremos en breve cómo *_steptrap* determina estas cosas; ahora veremos *_cmdloop*. Es un bucle de comandos típico, que se asemeja a una combinación de las declaraciones de caso que vimos en el Capítulo 5 y el bucle del calculador que vimos en el Capítulo 8.

```
# Bucle de comandos del depurador.
# Aquí al inicio de la sesión del depurador, cuando se alcanza un punto de interrupción,
# después de paso a paso. Opcionalmente aquí dentro de un punto de observación.
function _cmdloop {
  typeset cmd args
  while read -s cmd"?kshdb> " args; do
    case $cmd in
      \#bp )
        _setbp $args ;;         # establecer punto de interrupción en número de línea o cadena.
      \#bc )
        _setbc $args ;;         # establecer condición de interrupción.
      \#cb )
        _clearbp ;;             # borrar todos los puntos de interrupción.
      \#g )
        return ;;               # iniciar/reanudar ejecución
      \#s )
        let _steps=${args:-1}   # paso simple N veces (por defecto 1)
        return ;;
      \#wp )
        _setwp $args ;;         # establecer un punto de observación
      \#cw )
        _clearwp $args ;;       # borrar uno o más puntos de observación
      \#x )
        _xtrace ;;              # alternar rastreo de ejecución
      \#\? | \#h )
```

```
        _menu ;;                # imprimir menú de comandos
      \#q )
        exit ;;                 # salir
      \#* )
        _msg "Comando no válido: $cmd" ;;
      * )
        eval $cmd $args ;;      # de lo contrario, ejecutar comando de shell
    esac
  done
}
```

En cada iteración, *__cmdloop* imprime un indicador, lee un comando y lo procesa. Usamos `read -s` para que el usuario pueda aprovechar la edición de línea de comandos dentro de *kshdb*. Todos los comandos de *kshdb* comienzan con # para evitar confusiones con los comandos de shell. Cualquier cosa que no sea un comando de *kshdb* (y no comience con #) se pasa al shell para su ejecución. El uso de # como carácter de comando evita que un comando mal escrito tenga algún efecto perjudicial cuando la última declaración lo captura y lo ejecuta a través de *eval*. La Tabla 9.5 resume los comandos del depurador.

Tabla 9.5: Comandos de *kshdb*

Comando	**Acción**
#bp *N*	Establecer punto de interrupción en la línea N.
#bp *str*	Establecer punto de interrupción en la siguiente línea que contenga *str*.
#bp	Lista de puntos de interrupción y condición de interrupción.
#bc *str*	Establece la condición de rotura en *str*.
#bc	Condición de rotura clara.
#cb	Borra todos los puntos de interrupción.
#g	Iniciar o reanudar la ejecución (go).
#s *[N]*	Paso a través de *N* sentencias (por defecto 1).
#wp [-c] *var* get	Establece un watchpoint en la variable *var* cuando se recupera el valor. Con `-c`, entra en el bucle de comandos desde dentro del punto de control.
#wp [-c] *var* set	Establece un watchpoint en la variable *var* cuando se asigna el valor. Con `-c`, entra en el bucle de comandos desde dentro del punto de control.
#wp [-c] *var* unset	Establece un punto de control en la variable *var* cuando la variable no está establecida. Con `-c`, entra en el bucle de comandos desde dentro del punto de control.
#cw *var discipline*	Borra el punto de control dado.
#wc	Borra todos los puntos de vigilancia.
#x	Activar el seguimiento de la ejecución.
#h, #?	Imprime un menú de ayuda.
#q	Salir

Antes de examinar los comandos individuales, es importante que entiendas cómo pasa el

control a través de *_steptrap*, el bucle de comandos y el programa de pruebas.

_steptrap se ejecuta antes de cada declaración en el programa de pruebas como resultado de la declaración de `trap ... DEBUG` en el preámbulo. Si se ha alcanzado un punto de interrupción o el usuario escribió previamente un comando de paso (#s), *_steptrap* llama al bucle de comandos. Al hacerlo, interrumpe efectivamente el shell que está ejecutando el programa de pruebas para entregar el control al usuario. [6]

El usuario puede invocar comandos del depurador, así como comandos de shell que se ejecutan en el mismo shell que el programa de pruebas. Esto significa que puedes usar comandos de shell para verificar los valores de las variables, las trampas de señales y cualquier otra información local al script que se está depurando.

El bucle de comandos se ejecuta, y el usuario mantiene el control, hasta que el usuario escribe `#g, #s` o `#q`. Veamos en detalle qué sucede en cada uno de estos casos.

`#g` tiene el efecto de ejecutar el programa de pruebas sin interrupciones hasta que termine o alcance un punto de interrupción. Pero en realidad, simplemente sale del bucle de comandos y vuelve a *_steptrap*, que también sale. El shell retoma el control; ejecuta la siguiente declaración en el script del programa de pruebas y llama a *_steptrap* nuevamente. Suponiendo que no hay un punto de interrupción, esta vez *_steptrap* simplemente sale nuevamente, y el proceso se repite hasta que hay un punto de interrupción o el programa de pruebas ha terminado.

Paso a paso

Cuando el usuario escribe `#s`, el código del bucle de comandos establece la variable `_steps` en el número de pasos que el usuario desea ejecutar, es decir, en el argumento proporcionado. Supongamos en un principio que el usuario omite el argumento, lo que significa que `_steps` se establece en 1. Luego, el bucle de comandos sale y devuelve el control a *_steptrap*, que (como se mencionó anteriormente) sale y devuelve el control al shell. El shell ejecuta la siguiente declaración y vuelve a *_steptrap*, que ve que `_steps` es 1 y lo decrementa a 0. Luego, la tercera cláusula `elif` ve que `_steps` es 0, por lo que imprime un mensaje de «detenido» y llama al bucle de comandos.

Ahora supongamos que el usuario proporciona un argumento a #s, digamos 3. `_steps` se establece en 3. Entonces sucede lo siguiente:

[6] De hecho, los programadores de sistemas de bajo nivel pueden pensar en todo el mecanismo de *trampa* como bastante similar a un esquema de manejo de interrupciones.

1. Después de que se ejecuta la siguiente declaración, *__steptrap* se llama nuevamente. Entra en la primera cláusula `if`, ya que `_steps` es mayor que 0. *__steptrap* decrementa `_steps` a 2 y sale, devolviendo el control al shell.

2. Este proceso se repite, se ejecuta otro paso en el programa de pruebas, y `_steps` se convierte en 1.

3. Se ejecuta una tercera declaración y volvemos a *__steptrap*. `_steps` se decrementa a 0, se ejecuta la tercera cláusula `elif`, y *__steptrap* rompe de nuevo al bucle de comandos.

El efecto general es que se ejecutan tres pasos y luego el depurador toma el control nuevamente.

Finalmente, el comando #q sale. La trampa EXIT luego llama a la función *__cleanup*, que simplemente borra el archivo temporal y sale del programa completo.

Todos los demás comandos del depurador (#bp, #bc, #cb, #wp, #cw, #x, y comandos de shell) hacen que el shell permanezca en el bucle de comandos, lo que significa que el usuario prolonga la interrupción del shell.

Puntos de ruptura

Ahora examinaremos los comandos relacionados con los puntos de interrupción y el mecanismo de puntos de interrupción en general. El comando #bp llama a la función *__setbp*, que puede establecer dos tipos de puntos de interrupción, según el tipo de argumento proporcionado. Si es un número, se trata como un número de línea; de lo contrario, se interpreta como una cadena que debería contener la línea de punto de interrupción.

Por ejemplo, el comando `#bp 15` establece un punto de interrupción en la línea 15, y `#bp grep` establece un punto de interrupción en la siguiente línea que contiene la cadena *grep* –, sea cual sea el número que resulte. Aunque siempre puedes consultar una lista numerada de un archivo, [7]. los argumentos de cadena para #bp pueden hacer que esto no sea necesario.

Aquí está el código para *__setbp*:

```
# Establece punto(s) de interrupción en los números de línea o cadenas dados
# agregando patrones a las variables de punto de interrupción
function _setbp {
```

[7] `pr -n filename` imprime una lista numerada en la salida estándar en versiones derivadas de Unix basadas en System V. Algunos sistemas antiguos derivados de BSD no lo admiten. Si esto no funciona en tu sistema, prueba con `cat -n filename`, o si eso no funciona, crea un script de shell con la única línea `awk '{ printf("%d\t%s\n", NR, $0 }' $1`

```
  if [[ -z $1 ]]; then
    _listbp
  elif [[ $1 == +([0-9]) ]]; then                # número, establece punto de interrupción en
    esa línea
    _linebp="${_linebp}$1|"
    _msg "Punto de interrupción en la línea " $1
  else                                           # cadena, establece punto de interrupción en la
     siguiente línea con la cadena
    _stringbp="${_stringbp}$@|"
    _msg "Punto de interrupción en la próxima línea que contiene '$@'."
  fi
}
```

_setbp establece los puntos de interrupción almacenándolos en las variables `_linebp` (puntos de interrupción de número de línea) y `_stringbp` (puntos de interrupción de cadena). Ambos tienen puntos de interrupción separados por caracteres de tubería, por razones que se harán claras en breve. Esto implica que los puntos de interrupción son acumulativos; establecer nuevos puntos de interrupción no borra los antiguos.

La única forma de eliminar los puntos de interrupción es con el comando *#cb*, que (en la función *_clearbp*) los borra todos de una vez simplemente restableciendo las dos variables a nulo. Si no recuerdas qué puntos de interrupción has establecido, el comando #bp sin argumentos los enumera.

Las funciones *_at_linenumbp* y *_at_stringbp* son llamadas por *_steptrap* después de cada declaración; verifican si el shell ha llegado a un punto de interrupción de número de línea o de cadena, respectivamente.

Aquí está *_at_linenumbp*:

```
# Verifica si el próximo número de línea es un punto de interrupción.
function _at_linenumbp {
  [[ $_curline == @(${_linebp%\|}) ]]
}
```

_at_linenumbp aprovecha el carácter de tubería como el separador entre los números de línea: construye una expresión regular de la forma `(N1|N2|...)` tomando la lista de números de línea *_linebp*, eliminando el | final y rodeándolo con `(` y `)`. Por ejemplo, si `$_linebp` es `3|15|19|`, la expresión resultante es `(3|15|19)`.

Si la línea actual es cualquiera de estos números, la condición se vuelve verdadera, y *_at_linenumbp* también devuelve un estado de salida «verdadero» (0).

La verificación de un punto de interrupción de cadena funciona sobre el mismo principio, pero es ligeramente más complicada; aquí está *_at_stringbp*:

```
# Busca puntos de interrupción de cadena para ver si la próxima línea en el script coincide.
function _at_stringbp {
  [[ -n $_stringbp && ${_lines[$_curline]} == *@(${_stringbp%\|})* ]]
}
```

La condición primero verifica si `$_stringbp` no es nulo (lo que significa que se han definido puntos de interrupción de cadena). Si no es así, la condición se evalúa como falsa, pero si lo es, su valor depende de la coincidencia de patrones después del &&, que prueba la línea actual para ver si contiene alguna de las cadenas de puntos de interrupción.

La expresión en el lado derecho del signo igual doble es similar a la de *__at_linenumbp* anterior, excepto que tiene * antes y después. Esto da expresiones de la forma `*(S1|S2|...)*`, donde las S son los puntos de interrupción de cadena. Esta expresión coincide con cualquier línea que contenga alguna de las posibilidades entre paréntesis.

El lado izquierdo del signo igual doble es el texto de la línea actual en el programa de pruebas. Entonces, si este texto coincide con la expresión regular, hemos llegado a un punto de interrupción de cadena; en consecuencia, la expresión condicional y `_at_stringbp` devuelven un estado de salida 0.

__steptrap prueba cada condición por separado, para que pueda decirte qué tipo de punto de interrupción detuvo la ejecución. En ambos casos, llama al bucle principal de comandos.

9.2.4. Condiciones de interrupción

kshdb tiene otra característica relacionada con los puntos de interrupción: la *condición de interrupción*. Esta es una cadena que el usuario puede especificar y que se evalúa como un comando; si es verdadera (es decir, devuelve un estado de salida 0), el depurador entra en el bucle de comandos. Dado que la condición de interrupción puede ser cualquier línea de código de shell, hay mucha flexibilidad en lo que se puede probar. Por ejemplo, puedes interrumpir cuando una variable alcanza un cierto valor (por ejemplo, `(( $x < 0 ))`) o cuando un texto en particular se ha escrito a un archivo (`grep string archivo`). Probablemente pensarás en todo tipo de usos para esta característica. [8] Para establecer una condición de interrupción, escribe `#bc string`. Para eliminarla, escribe `#bc` sin argumentos; esto instala la cadena nula, que se ignora. *__steptrap* evalúa la condición de interrupción `$_brcond` solo si no es nula. Si la condición de interrupción se evalúa como 0, la cláusula `if` es verdadera

[8] Ten en cuenta que si tu condición de interrupción produce alguna salida estándar (o error estándar), la verás antes de cada declaración. Además, asegúrate de que tu condición de interrupción no tarde mucho tiempo en ejecutarse; de lo contrario, tu script se ejecutará muy, muy lentamente.

y, una vez más, *_steptrap* llama al bucle de comandos.

Rastreo de ejecución

La siguiente característica es el rastreo de ejecución, disponible a través del comando `#x`. Esta función está diseñada para superar el hecho de que un usuario de *kshdb* no puede usar `set -o xtrace` mientras depura (ingresándolo como un comando de shell), porque su alcance se limita a la función *_cmdloop*. [9]

La función *_xtrace* alterna el rastreo de ejecución simplemente asignando a la variable *_trace* la negación lógica de su valor actual, de modo que alterna entre 0 (apagado) y 1 (encendido). La preámbulo lo inicializa en 0.

Puntos de observación

kshdb aprovecha las funciones de disciplina del shell para proporcionar puntos de observación. Puedes establecer un punto de observación en cualquier variable cuando se recupera o cambia el valor de la variable, o cuando la variable se anula. Opcionalmente, el punto de observación se puede configurar para ingresar también al bucle de comandos. Haces esto con el comando #wp, que a su vez llama a *_setwp*:

```
# Establece un punto de observación en una variable
# uso: _setwp [-c] var disciplina
# $1 = variable
# $2 = get|set|unset
typeset -A _watchpoints
function _setwp {
  typeset funcdef do_cmdloop=0
  if [[ $1 == -c ]]; then
    do_cmdloop=1
    shift
  fi

  funcdef="function $1.$2 { "

  case $2 in
    get)      funcdef+="_msg $1 \(\$$1\) recuperado, línea \$_curline"
              ;;
    set)      funcdef+="_msg $1 configurado a "'${.sh.value}'", línea \$_curline"
              ;;
    unset)    funcdef+="_msg $1 anulado en línea \$_curline"
              funcdef+=$'\nunset '"$1"
              ;;
    *) _msg   función de punto de observación $2 no válida
```

[9] De hecho, ingresando `typeset -ft funcname`, el usuario puede habilitar el rastreo de manera específica por función, pero probablemente sea mejor tenerlo todo bajo el control del depurador.

```
                return 1
                ;;
  esac

  if ((do_cmdloop)); then
    funcdef+=$'\n_cmdloop'
  fi
  funcdef+=$'\n}'

  eval "$funcdef"

  _watchpoints[$1.$2]=1
}
```

Esta función ilustra varias técnicas interesantes. Lo primero que hace es declarar algunas variables locales y verificar si se invocó con la opción `-c`. Esto indica que el punto de observación debería ingresar al bucle de comandos.

La idea general es construir el texto de la función de disciplina apropiada en la variable `funcdef`. El valor inicial es la palabra clave `function`, el nombre de la función de disciplina y la llave izquierda de apertura. El espacio después de la llave es importante, para que el shell la reconozca correctamente como una palabra clave.

Luego, para cada tipo de función de disciplina, el bloque `case` agrega el cuerpo de función apropiado a la cadena `funcdef`. El código utiliza barras invertidas colocadas de manera prudente para obtener la mezcla correcta de evaluación inmediata y diferida de variables del shell. Considera el caso `get`: para `\(`, la barra invertida se mantiene intacta para usarla como un carácter de cotización dentro del cuerpo de la función de disciplina. Para `\$$1`, la cotización sucede de la siguiente manera: el `\$` se convierte en un $ dentro de la función, mientras que el $1 se evalúa inmediatamente dentro de la cadena entre comillas dobles.

En el caso de que se haya suministrado la opción `-c`, utiliza la notación `$'...'` para agregar una nueva línea y llamar a *_cmdloop* al cuerpo de la función, y luego al final agrega otra nueva línea y una llave derecha de cierre. Finalmente, mediante *eval*, instala la función recién creada.

Por ejemplo, si se usó `-c`, el texto de la función *get* generada para la variable `count` termina viéndose así:

```
function count.get {
  _msg count \($count\) recuperado, línea $_curline
  _cmdloop
}
```

Al final de *_setwp*, `_watchpoints[$1.$2]` se establece en 1. Esto crea una entrada en el array asociativo `_watchpoints` indexada por el nombre de la función de disciplina. Esto almacena convenientemente los nombres de todos los puntos de observación para cuando queramos borrarlos.

Los puntos de observación se borran con el comando `#cw`, que a su vez ejecuta la función *_clearwp*. Aquí está:

```
# Borrar puntos de observación:
# sin argumentos: borrar todos
# dos argumentos: igual que para establecer: var get|set|unset
function _clearwp {
  if [ $# = 0 ]; then
    typeset _i
    for _i in ${!_watchpoints[*]}; do
      unset -f $_i
      unset _watchpoints[$_i]
    done
  elif [ $# = 2 ]; then
    case $2 in get | set | unset)
      unset -f $1.$2
      unset _watchpoints[$1.$2] ;;
    *) _msg $2: punto de observación no válido ;;
    esac
  fi
}
```

Cuando se invoca sin argumentos, *_clearwp* borra todos los puntos de observación, recorriendo todos los subíndices en el array asociativo `_watchpoints`. De lo contrario, si se invoca con dos argumentos, el nombre de la variable y la función de disciplina, anula la función utilizando `unset -f`. En ambos casos, también se anula la entrada en `_watchpoints`.

Limitaciones

kshdb no fue diseñado para llevar el estado del arte del depurador hacia adelante o tener un exceso de funciones. Tiene las características básicas más útiles; su implementación es compacta y (esperamos) comprensible. Pero tiene algunas limitaciones importantes. Las que conocemos se describen en la lista que sigue:

- Los puntos de interrupción de cadena no pueden comenzar con dígitos ni contener caracteres de tubería (|) a menos que estén correctamente escapados.
- Solo puedes establecer puntos de interrupción, ya sea de número de línea o de cadena, en líneas del programa de pruebas que contienen lo que la documentación del shell llama *comandos simples*, es decir, comandos Unix reales, integrados en el shell,

llamadas a funciones o alias. Si estableces un punto de interrupción en una línea que contiene solo espacios en blanco o un comentario, el shell siempre omitirá ese punto de interrupción. Más importante aún, las palabras clave de control como `while, if, for, do, done` e incluso condicionales (`[[...]]` y `((...))`) tampoco funcionarán, a menos que haya un comando simple en la misma línea.

- *kshdb* no «desciende> a los scripts de shell que se llaman desde el programa de pruebas. Para hacer esto, debes editarlo y cambiar una llamada a *scriptname* a `kshdb scriptname`.
- De manera similar, las subcáscaras se tratan como una sola declaración gigantesca; no puedes descender en ellas en absoluto.
- El programa de pruebas no debe atrapar las señales ficticias DEBUG o EXIT; de lo contrario, el depurador no funcionará.
- Las variables que están *tipadas* (ver Capítulo 4) no son accesibles en condiciones de interrupción. Sin embargo, puedes usar el comando del shell *print* para verificar sus valores.
- El manejo de errores de comando es débil. Por ejemplo, un argumento no numérico para `#s` hará que falle.
- Los puntos de observación que invocan al bucle de comandos son frágiles. Para *ksh93m* en GNU/Linux, intentar anular un punto de observación cuando está en el bucle de comandos invocado desde el punto de observación hace que el shell genere un volcado de núcleo. Pero esto no sucede en todas las plataformas, y esto se solucionará eventualmente.

Muchos de estos problemas no son insuperables; consulta los ejercicios.

9.2.5. Ejemplo de sesión kshdb

A continuación, mostraremos una transcripción de una sesión real con *kshdb*, en el que el programa de pruebas es (una versión ligeramente modificada) de la solución para la Tarea 6-3. Para mayor comodidad, aquí tienes una lista numerada del script, que llamaremos *lscol*.

```
set -A filenames $(ls $1)
typeset -L14 fname
let numfiles=${#filenames[*]}
let numcols=5
```

```
for ((count = 0; $count < $numfiles ; )); do

  fname=${filenames[count]}
  print -n "$fname "
  let count++
  if (( count % numcols == 0 )); then
    print # nueva línea
  fi
done

if (( count % numcols != 0 )); then
  print
fi
```

Aquí está la transcripción de la sesión de *kshdb*:

```
$ kshdb lscol book
Korn Shell Debugger version 2.0 for ksh Version M 1993-12-28 m
Stopped at line 1
kshdb> #bp 4
Breakpoint at line 4
kshdb> #g
Reached line breakpoint at line 4
kshdb> #s
Stopped at line 6
kshdb> print $numcols
5
kshdb> #bc (( count == 10 ))
Break when true: (( count == 10 ))
kshdb> #g
appa.xml      appb.xml      appc.xml      appd.xml      appf.xml
book.xml      ch00.xml      ch01.xml      ch02.xml      ch03.xml
Break condition '(( count == 10 ))' true at line 10
kshdb> #bc
Break condition cleared.
kshdb> #bp newline
Breakpoint at next line containing 'newline'.
kshdb> #g
Reached string breakpoint at line 11
kshdb> print $count
10
kshdb> let count=9
kshdb> #g

ch03.xml      Reached string breakpoint at line 11
kshdb> #bp
Breakpoints at lines:
4
Breakpoints at strings:
newline
Break on condition:
```

```
kshdb> #g

ch04.xml      ch05.xml      ch06.xml      ch07.xml      ch08.xml
Reached string breakpoint at line 11
kshdb> #g

ch09.xml      ch10.xml      colo1.xml      copy.xml
$
```

Primero, observa que le dimos al script de pruebas el argumento `book`, lo que significa que queremos listar los archivos en ese directorio. Comenzamos estableciendo un punto de interrupción simple en la línea 4 y ejecutamos el script. Se detiene antes de ejecutar la línea 4 (`let numcols=5`). Emitimos el comando `#s` para ejecutar paso a paso el comando (es decir, para ejecutarlo realmente). Luego emitimos un comando *print* del shell para mostrar que la variable `numcols` está configurada correctamente.

A continuación, establecemos una condición de interrupción, indicándole al depurador que intervenga cuando `count` sea 10, y reanudamos la ejecución. Efectivamente, el programa de pruebas imprime 10 nombres de archivo y se detiene en la línea 10, justo después de incrementar `count`. Borramos la condición de interrupción escribiendo `#bc` sin un argumento, ya que de lo contrario, el shell se detendría después de cada declaración hasta que la condición se volviera falsa.

El siguiente comando muestra cómo funciona el mecanismo de punto de interrupción de cadena. Le decimos al depurador que interrumpa cuando llegue a una línea que contenga la cadena `newline`. Esta cadena está en un comentario en la línea 11. Observa que no importa que la cadena esté en un comentario, solo importa que la línea en la que se encuentra contenga un comando real. Continuamos la ejecución y el depurador alcanza el punto de interrupción en la línea 11.

Después de eso, mostramos cómo podemos usar el depurador para cambiar el estado del programa de pruebas mientras se ejecuta. Vemos que `$count` sigue siendo mayor que 10; lo cambiamos a 9. En la siguiente iteración del bucle `while`, el script accede al mismo nombre de archivo que acaba de hacerlo (*ch03.xml*), incrementa `count` de nuevo a 10 y alcanza nuevamente el punto de interrupción de cadena. Finalmente, listamos los puntos de interrupción y avanzamos hasta el final, momento en el que sale.

9.2.6. Ejercicios

Concluimos este capítulo con algunos ejercicios, que son mejoras sugeridas para *kshdb*.

1. Mejora el manejo de errores de comandos de las siguientes maneras:

   a) Para argumentos numéricos de `#bp`, verifica que sean números de línea válidos para el programa de pruebas en particular.

   b) Verifica que los argumentos de `#s` sean números válidos.

   c) Cualquier otro manejo de errores que se te ocurra.

2. Mejora el comando `#cb` para que el usuario pueda eliminar puntos de interrupción específicos (por cadena o número de línea).

3. Elimina la principal limitación en el mecanismo de puntos de interrupción:

   a) Mejóralo de manera que si el número de línea seleccionado no contiene un comando Unix real, se utilice en su lugar la línea más cercana por encima de ella como punto de interrupción.

   b) Haz lo mismo para los puntos de interrupción de cadena. (Pista: primero traduce cada comando de punto de interrupción de cadena en uno o más comandos de punto de interrupción por número de línea).

4. Implementa una opción que provoque una interrupción en el depurador cada vez que un comando sale con un estado distinto de cero:

   a) Impleméntalo como la opción de línea de comandos `-e`.

   b) Impleméntalo como los comandos del depurador `#be` (para activar la opción) y `#ne` (para desactivarla). (Pista: no podrás usar el *trap ERR*, pero ten en cuenta que cuando entras en *_steptrap*, `$?` sigue siendo el estado de salida del último comando que se ejecutó).

5. Añade la capacidad de «descender» a los scripts que en los que el programa de pruebas llama (es decir, subprocesos de shell) como la opción de línea de comandos `-s`. Una forma de implementar esto es cambiar el script *kshdb* para que inserte llamadas recursivas a *kshdb* en el programa de pruebas. Puedes hacer esto filtrando el programa de pruebas a través de un bucle que lee cada línea y determina, con los comandos `whence -v` y *file(1)* (consulta la página de manual), si la línea es una llamada a otro script de shell. [10] Si es así, antepón `kshdb -s` a la línea y escríbela en el nuevo archivo; si no, simplemente pásala tal como está.

[10] Ten en cuenta que este método debería capturar la mayoría de los scripts de shell separados, pero no todos. Por ejemplo, no capturará scripts de shell que sigan a punto y coma (por ejemplo, `cmd1; cmd2`).

6. Añade soporte para múltiples condiciones de interrupción, de modo que *kshdb* detenga la ejecución cuando cualquiera de ellas sea verdadera e imprima un mensaje que indique cuál es verdadera. Haz esto almacenando las condiciones de interrupción en una lista separada por dos puntos o en un array. Intenta que esto sea lo más eficiente posible, ya que la verificación debe realizarse antes de cada declaración.

7. Añade cualquier otra característica que se te ocurra.

Finalmente, aquí tienes el código fuente completo para el archivo de funciones del depurador *kshdb.fns*:

```
# Aquí antes de cada declaración en el script en depuración.
# Maneja el paso individual y los puntos de interrupción.
function _steptrap {
  _curline=$1                       # el argumento es el número de línea que acaba de ejecutarse
  (( $_trace )) && _msg "$PS4 línea $_curline: ${_lines[$_curline]}"
  if (( $_steps >= 0 )); then       # si está en modo paso a paso
    let _steps="$_steps - 1"        # decrementa el contador
  fi

  # primera comprobación: si se alcanza un punto de interrupción por número de línea
  if _at_linenumbp; then
    _msg "Se alcanzó un punto de interrupción en la línea $_curline"
    _cmdloop                        # punto de interrupción, entra en el depurador

  # segunda comprobación: si se alcanza un punto de interrupción por cadena
  elif _at_stringbp; then
    _msg "Se alcanzó un punto de interrupción de cadena en la línea $_curline"
    _cmdloop                        # punto de interrupción, entra en el depurador

  # si no, comprueba si existe una condición de interrupción y es verdadera
  elif [[ -n $_brcond ]] && eval $_brcond; then
    _msg "La condición de interrupción '$_brcond' es verdadera en la línea $_curline"
    _cmdloop                        # condición de interrupción, entra en el depurador

  # finalmente, comprueba si está en modo paso a paso y el número de pasos ha terminado
  elif (( _steps == 0 )); then      # si está en modo paso a paso y es hora de detenerse
    _msg "Detenido en la línea $_curline"
    _cmdloop                        # entra en el depurador
  fi
}

# Bucle de comandos del depurador.
# Aquí al inicio de la sesión del depurador, cuando se alcanza un punto de interrupción,
# después de un paso individual. Opcionalmente aquí dentro de un punto de control.
function _cmdloop {
  typeset cmd args

  while read -s cmd"?kshdb> " args; do
    case $cmd in
```

```
      \#bp ) _setbp $args ;;        # establece un punto de interrupción en el número de línea o
     cadena.
      \#bc ) _setbc $args ;;        # establece una condición de interrupción.
      \#cb ) _clearbp ;;            # elimina todos los puntos de interrupción.
      \#g ) return ;;               # iniciar/reanudar la ejecución.
      \#s ) let _steps=${args:-1}   # ejecutar N declaraciones una vez (predeterminado 1).
            return ;;
      \#wp ) _setwp $args ;;        # establece un punto de control en una variable.
      \#cw ) _clearwp $args ;;      # elimina uno o más puntos de control.
      \#x ) _xtrace ;;              # alternar seguimiento de ejecución on/off.
      \#\? | \#h ) _menu ;;         # imprimir menú de comandos.
      \#q ) exit ;;                 # salir.
      \#* ) _msg "Comando no válido: $cmd" ;;
      * ) eval $cmd $args ;;        # de lo contrario, ejecuta el comando de shell.
    esac
  done
}

# Verifica si el próximo número de línea es un punto de interrupción.
function _at_linenumbp {
  [[ $_curline == @(${_linebp%\|}) ]]
}

# Busca puntos de interrupción de cadena para ver si la siguiente línea del script coincide.
function _at_stringbp {
  [[ -n $_stringbp && ${_lines[$_curline]} == *@(${_stringbp%\|})* ]]
}

# Imprime el mensaje dado en la salida de error estándar.
function _msg {
  print -r -- "$@" >&2
}

# Establece punto(s) de interrupción en números de línea o cadenas
# mediante la adición de patrones a las variables de puntos de interrupción.
function _setbp {
  if [[ -z $1 ]]; then
    _listbp
  elif [[ $1 == +([0-9]) ]]; then     # número, establece un punto de interrupción en esa línea
    _linebp="${_linebp}$1|"
    _msg "Punto de interrupción en la línea " $1
  else                                # string, establece un punto de interrupción en la
    siguiente línea con el string
    _stringbp="${_stringbp}$@|"
    _msg "Punto de interrupción en la siguiente línea que contiene '$@'."
  fi
}

# Lista puntos de interrupción y condición de interrupción
function _listbp {
  _msg "Puntos de interrupción en líneas:"
  _msg "$(print $_linebp | tr '|' ' ')"
```

```
  _msg "Puntos de interrupción en cadenas:"
  _msg "$(print $_stringbp | tr '|' ' ')"
  _msg "Interrupción en condición:"
  _msg "$_brcond"
}

# Establece o elimina la condición de interrupción.
function _setbc {
  if [[ $# = 0 ]]; then
    _brcond=
    _msg "Condición de interrupción eliminada."
  else
    _brcond="$*"
    _msg "Interrumpir cuando sea verdadero: $_brcond"
  fi
}

# Elimina todos los puntos de interrupción.
function _clearbp {
  _linebp=
  _stringbp=
  _msg "Todos los puntos de interrupción eliminados."
}

# Alterna la función de seguimiento de ejecución on/off.
function _xtrace {
  let _trace="! $_trace"
  if (( $_trace )); then
    _msg "Seguimiento de ejecución activado."
  else
    _msg "Seguimiento de ejecución desactivado."
  fi
}

# Imprime menú de comandos.
function _menu {
  _msg 'Comandos kshdb:
  #bp N                        establece un punto de interrupción en la línea N
  #bp str                      establece un punto de interrupción en la siguiente línea que
    contiene str
  #bp                          lista puntos de interrupción y condición de interrupción
  #bc str                      establece la condición de interrupción a str
  #bc                          elimina la condición de interrupción
  #cb                          elimina todos los puntos de interrupción
  #wp [-c] var discipline      establece un punto de control en una variable
  #cw                          elimina todos los puntos de control
  #g                           inicia/reanuda la ejecución
  #s [N]                       ejecuta N declaraciones (predeterminado 1)
  #x                           alterna el seguimiento de ejecución on/off
  #h, #?                       imprime este menú
  #q                           salir'
}
```

```
# Borra archivos temporales antes de salir.
function _cleanup {
  rm $_dbgfile 2>/dev/null
}

# Establece un punto de control en una variable
# uso: _setwp [-c] var disciplina
# $1 = variable
# $2 = get|set|unset
typeset -A _watchpoints
function _setwp {
  typeset funcdef do_cmdloop=0
  if [[ $1 == -c ]]; then
    do_cmdloop=1
    shift
  fi

  funcdef="function $1.$2 { "
  case $2 in
    get)    funcdef+="_msg $1 \(\$$1\) obtenido, línea \$_curline"
            ;;
    set)    funcdef+="_msg $1 establecido a "'${.sh.value}'", línea \$_curline"
            ;;
    unset)  funcdef+="_msg $1 eliminado en la línea \$_curline"
            funcdef+=$'\nunset '"$1"
            ;;
    *)      _msg función de punto de control no válida $2
            return 1
            ;;
  esac

  if ((do_cmdloop)); then
    funcdef+=$'\n_cmdloop'
  fi
  funcdef+=$'\n}'

  eval "$funcdef"

  _watchpoints[$1.$2]=1
}

# Elimina puntos de control:
# sin argumentos, elimina todos
# con dos argumentos: lo mismo que para el establecimiento: var get|set|unset
function _clearwp {
  if [ $# = 0 ]; then
    typeset _i
    for _i in ${!_watchpoints[*]}; do
      unset -f $_i
      unset _watchpoints[$_i]
    done
```

```
  elif [ $# = 2 ]; then
    case $2 in
      get | set | unset)
          unset -f $1.$2
          unset _watchpoints[$1.$2]
          ;;
      *) _msg $2: punto de control no válido
          ;;
    esac
  fi
}
```

CAPÍTULO 10

ADMINISTRACIÓN DE KORN SHELL

Los administradores de sistemas utilizan el intérprete de comandos como parte de su trabajo de configurar un entorno en todo el sistema para todos los usuarios. En este capítulo, discutiremos las características del shell Korn que se relacionan con esta tarea desde dos perspectivas: la personalización que está disponible para todos los usuarios y la seguridad del sistema. Asumimos que ya conoce los fundamentos de la administración de sistemas Unix.[1]

10.1. Instalación del shell Korn como shell estándar

Como preludio a la personalización de todo el sistema, queremos destacar algo sobre el shell Korn que no se aplica a la mayoría de los otros shells: puede instalarlo como si fuera el shell Bourne estándar, es decir, como `/bin/sh`. Simplemente guarde el shell Bourne real como otro nombre de archivo, como `/bin/bsh`, en caso de que alguien realmente lo necesite para algo (lo cual es dudoso), luego renombre (o enlace) su shell Korn como `/bin/sh`.

Muchas instalaciones han hecho esto sin ningún efecto negativo. Esto no sólo hace que el shell Korn sea el shell de inicio de sesión estándar de su sistema, sino que también hace que la mayoría de los scripts del shell Bourne existentes se ejecuten más rápido, y tiene ventajas de seguridad que veremos más adelante en este capítulo.

Como veremos en el Apéndice A, el shell Korn es retrocompatible con el shell Bourne excepto que no soporta ^ como sinónimo del carácter de tubería |. A menos que tenga un sistema Unix antiguo, o tenga algunos scripts de shell muy, muy antiguos, no necesita preocuparse por esto.

[1] Una buena fuente de información sobre administración de sistemas es *Essential System Administration* de Æleen Frisch. Está publicado por O'Reilly & Associates.

Pero si quiere estar absolutamente seguro, simplemente busque entre todos los scripts de shell en todos los directorios de su `PATH`. Una manera fácil de hacer esto es usar el comando *file(1)*, que vimos en el Capítulo 5 y en el Capítulo 9. *file* imprime «script de shell ejecutable» cuando se le da el nombre de uno. (El mensaje exacto varía de un sistema a otro; asegúrese de que el suyo imprime este mensaje cuando se le da el nombre de un script de shell. Si no es así, simplemente sustituya el mensaje que imprime su comando *file* por «shell script» en el siguiente ejemplo). He aquí un script que busca ^ en scripts de shell en cada directorio de su `PATH`:[2]

```
dirs=$(print -- $PATH |
  sed -e 's/^:/.:/' -e 's/::/:.:/' -e s'/:$/:./' -e 's/:/ /g')
for d in $dirs
do
  print "checking $d:"
  cd "$d"
  scripts=$(file * | grep 'shell script' | cut -d: -f1)
  grep -l '\^' $scripts /dev/null
done
```

La primera sentencia de este script separa `$PATH` en directorios separados, incluyendo el manejo de varios casos de separadores vacíos que significan el directorio actual. El programa *sed(1)* es un editor de flujo que realiza operaciones de edición en su entrada, e imprime los contenidos cambiados en su salida. El resultado se asigna a *dirs*, que se utiliza como lista de elementos en el bucle *for*. Para cada directorio, *cd*s allí y encuentra todos los scripts de shell mediante la canalización del comando *file* en *grep* y luego, para extraer sólo el nombre de archivo, en *cut*. A continuación, busca el carácter ^ en cada script. La opción `-l` de *grep* simplemente lista todos los nombres de archivo que coinciden con el patrón, sin imprimir las líneas coincidentes. El comando *grep* tiene */dev/null* al final de la lista de archivos en caso de que `$scripts` esté vacío. Si eres aventurero, puedes hacer todo el trabajo en una sola línea:

```
grep -l '\^' $(file * | grep 'shell script' | cut -d: -f1) /dev/null
```

Si ejecutas este script, probablemente encontrarás varias apariciones de ^ pero éstas deberían ser usadas dentro de expresiones regulares en comandos *grep, sed,* o *awk*, no como caracteres de tubería. Asumiendo que este es el caso, es seguro que instale el shell Korn como */bin/sh*.

[2]Este script fallará si su PATH tiene directorios cuyos nombres contienen espacios. Considere solucionar este problema como un ejercicio avanzado para el lector.

10.2. Personalización del entorno

Al igual que el shell de Bourne, el shell de Korn utiliza el archivo `/etc/profile` para la personalización a nivel del sistema. Cuando un usuario inicia sesión, el shell lee y ejecuta `/etc/profile` antes de ejecutar el *.profile* del usuario.

No cubrimos todos los comandos posibles que podría querer colocar en `/etc/profile`. Pero el shell de Korn tiene algunas características únicas que son particularmente relevantes para la personalización a nivel del sistema; las discutimos aquí.

Comenzaremos con dos comandos integrados que puede utilizar en `/etc/profile` para adaptar los entornos de sus usuarios y limitar su uso de los recursos del sistema. Los usuarios también pueden usar estos comandos en su *.profile*, o en cualquier otro momento, para anular la configuración predeterminada.

10.2.1. umask

umask, al igual que el mismo comando en la mayoría de las otras shells, le permite especificar los permisos predeterminados que tienen los archivos cuando los usuarios los crean. Con *ksh*, toma los mismos tipos de argumentos que el comando *chmod*, es decir, valores de permisos absolutos (números octales) o simbólicos.

umask contiene los permisos que se desactivan de manera predeterminada cada vez que un proceso crea un archivo, independientemente de los permisos que especifique el proceso. [3] Otra forma de pensar en esto es como una sustracción libre de préstamos bit a bit: *permisos reales = permisos solicitados - umask*.

Usaremos la notación octal para mostrar cómo funciona esto. Como debería saber, los dígitos en un número de permiso representan (de izquierda a derecha) los permisos del propietario, el grupo del propietario y todos los demás usuarios, respectivamente. Cada dígito, a su vez, consta de tres bits, que especifican permisos de lectura, escritura y ejecución de izquierda a derecha. (Si un archivo es un directorio, el permiso «ejecutar» se convierte en permiso «buscar», es decir, permiso para *cd* a él y recorrerlo como parte de una ruta de acceso).

Por ejemplo, el número octal 640 es igual al número binario 110 100 000. Si un archivo tiene este permiso, entonces su propietario puede leerlo y escribirlo; los usuarios en el grupo

[3] Si conoce C, C++ o Java, y se siente cómodo con las operaciones bit a bit, la operación de *umask* funciona así: `permiso_real = permiso_solicitado & (~ umask)`.

del propietario solo pueden leerlo; los demás no tienen permisos sobre él. Un archivo con permiso 755 (111 101 101 en binario) le da a su propietario el derecho de leer, escribir y ejecutarlo y a todos los demás el derecho de leer y ejecutar (pero no escribir).

022 es un valor de *umask* común. Esto implica que cuando se crea un archivo, el permiso máximo que podría tener es 755, que es el permiso habitual de un ejecutable que un compilador podría crear. Un editor de texto, por otro lado, podría crear un archivo con permiso 666 (lectura y escritura para todos), pero la *umask* obliga a que sea 644 en su lugar.

La opción `-S` para *umask* hace que funcione con valores simbólicos en lugar de con números octales. Cuando se usa sin un argumento, `umask -S` imprime *umask* en forma simbólica. Con un argumento, se cambia la máscara. En ambos casos, una máscara simbólica representa los permisos que se deben *mantener* para un archivo. (Esto termina siendo el complemento bit a bit del umask octal tradicional, que representa permisos a eliminar). Si está confundido, algunos ejemplos deberían aclarar las cosas:

```
$ umask  # Imprime la umask actual, en octal
0022
$ umask -S               # Imprime la umask actual, en forma simbólica
u=rwx,g=rx,o=rx
$ umask -S u=rwx,g=r,o=  # Cambia la umask usando la forma simbólica
$ umask -S               # Imprímelo nuevamente de manera simbólica
u=rwx,g=r,o=
$ umask                  # Imprímelo en octal
0037
```

ulimit

En los primeros sistemas Unix, no se imponían límites a los recursos que un proceso podía utilizar. Si un programa quería ejecutarse para siempre, podía hacerlo. Si un programa quería crear archivos grandes y llenar un disco, podía hacerlo. Y así sucesivamente.

A medida que Unix se desarrolló y maduró, se hizo posible controlar explícitamente o *limitar* una variedad de recursos del sistema, como el tiempo de CPU y el espacio en disco. El comando *ulimit* es la interfaz del shell de Korn para ver y cambiar los límites de los recursos del sistema. La Tabla 10.1 enumera las opciones que acepta y los recursos correspondientes. No todas las opciones están disponibles en todos los sistemas Unix. Muchas no estarán disponibles en sistemas no Unix.

[4]La mayoría de los sistemas Unix no tienen esta característica.

Tabla 10.1: Opciones de recursos `ulimit`

Opción	**Recursos limitados**	**Opción**	**Recursos limitados**
-a	Todos (sólo para imprimir valores)	-n	Descriptores de archivos
-c	Tamaño del archivo principal ($\frac{1}{2}$ bloques de kb)	-p	Tamaño del búfer de la tubería ($\frac{1}{2}$ bloques de kb) [4]
-d	Segmento de datos de proceso (kb)	-s	Segmento de pila de proceso (kb)
-f	Tamaño del fichero ($\frac{1}{2}$ kb bloques)	-t	Tamaño del fichero (kb bloques)
-m	Memoria física (kb)	-v	Memoria física (kb)

Cada opción toma un argumento numérico que especifica el límite en las unidades que se muestran en la tabla. (Puede usar una expresión aritmética para el límite; *ksh* evalúa automáticamente la expresión). También puede dar el argumento «ilimitado» (que en realidad puede significar algún límite físico), o puede omitir el argumento, en cuyo caso imprime el límite actual. `ulimit -a` imprime los límites (o «ilimitado») para todos los tipos. Solo puede especificar un tipo de recurso a la vez. Si no especifica ninguna opción, se asume `-f`.

Algunas de estas opciones dependen de capacidades del sistema operativo que no existen en versiones antiguas de Unix. En particular, algunas versiones antiguas tienen un límite fijo de 20 descriptores de archivos por proceso (haciendo que `-n` sea irrelevante), y algunas no admiten memoria virtual (haciendo que `-v` sea irrelevante).

Las opciones `-d` y `-s` tienen que ver con la *asignación de memoria dinámica*, es decir, la memoria que un proceso solicita al sistema operativo en tiempo de ejecución. No es necesario que los usuarios ocasionales limiten estos, aunque los desarrolladores de software pueden querer hacerlo para evitar que programas con errores intenten asignar cantidades interminables de memoria debido a bucles infinitos.

La opción `-v` es similar; pone un límite a todos los usos de memoria. No necesita esto a menos que su sistema tenga restricciones severas de memoria o desee limitar el tamaño del proceso para evitar la fragmentación.

Es posible que desee especificar límites en el tamaño de los archivos (`-f` y `-c`) si tiene restricciones de espacio en disco. A veces, los usuarios realmente quieren crear archivos enormes, pero más a menudo, un archivo enorme es el resultado de un programa con errores que entra en un bucle infinito. Los desarrolladores de software que usan depuradores como *gdb* y *dbx* no deben limitar el tamaño del archivo central, porque los volcados centrales a menudo son útiles para la depuración.

La opción `-t` es otra posible protección contra bucles infinitos. En sistemas de un solo usuario, un programa que está en un bucle infinito pero no está asignando memoria, escribiendo archivos ni utilizando la red no es particularmente peligroso; es mejor dejar esto ilimitado y simplemente permitir que el usuario detenga el programa ofensor. Sin embargo, en sistemas de servidores compartidos, tales programas degradan definitivamente el entorno general. El problema en ese caso es que es difícil saber qué límite establecer: hay usos importantes y legítimos para programas de larga duración.

Además de los tipos de recursos que puede limitar, *ulimit* le permite especificar límites duros o blandos. Los límites duros pueden disminuirlos cualquier usuario, pero solo aumentarlos el superusuario (`root`); los usuarios pueden disminuir los límites blandos y aumentarlos, pero solo hasta el límite duro para ese recurso.

Si proporciona `-H` junto con una (o más) de las opciones anteriores, *ulimit* establece límites duros; `-S` establece límites blandos. Sin ninguno de estos, *ulimit* establece ambos. Por ejemplo, los siguientes comandos establecen el límite blando en descriptores de archivos en 64 y el límite duro en ilimitado:

```
ulimit -Sn 64
ulimit -Hn unlimited
```

Cuando *ulimit* imprime los límites actuales, imprime los límites blandos a menos que especifique `-H`.

10.2.2. Tipos de personalización global

El enfoque más adecuado para la personalización global disponible sería un archivo de entorno a nivel del sistema que sea independiente del archivo de entorno de cada usuario, al igual que `/etc/profile` es independiente del *.profile* de cada usuario.

Desafortunadamente, el shell de Korn no tiene esta característica. Si asigna un nombre de archivo a la variable de entorno `ENV`, podría anularse en el *.profile* de un usuario. Esto le permite proporcionar un archivo de entorno predeterminado para los usuarios que no tienen el suyo propio, pero no le permite tener un archivo de entorno a nivel del sistema que se ejecute además de los de los usuarios. Además, el archivo de entorno solo se ejecuta para shells interactivas, no para todas las shells.

Sin embargo, el shell le proporciona algunas formas de configurar personalizaciones que están disponibles para todos los usuarios en todo momento. Las variables de entorno son

las más obvias; su archivo `/etc/profile` seguramente contendrá definiciones para varias de ellas, incluyendo `PATH` y `TERM`.

La variable `TMOUT` es útil cuando su sistema admite líneas de conexión telefónica. Ya hemos visto que afecta al comando *read* y al bucle del menú `select`. Cuando se establece en un número *N*, si un usuario no ingresa un comando dentro de *N* segundos después de que el shell emitió por última vez un indicador, el shell imprime el mensaje de advertencia «el shell se cerrará en 60 segundos debido a inactividad». Si, después de otros 60 segundos, el usuario no ingresa nada, el shell se cierra. Esta función es útil para evitar que las personas «acaparen» las líneas telefónicas. ¡Solo asegúrese de establecerlo en un valor razonable!

Es posible que desee incluir algunas personalizaciones más complejas que involucren variables de entorno, como la cadena de indicadores PS1 que contiene el directorio actual, el nombre de usuario o el nombre del host (como se ve en el Capítulo 4).

También puede activar opciones, como los modos de edición *emacs* o *vi*, *noclobber* para protegerse contra la sobrescritura inadvertida de archivos y tal vez *ignoreeof* para evitar que las personas cierren la sesión accidentalmente cuando escriben demasiados caracteres CTRL-D. Cualquier script de shell que haya escrito para uso general también contribuye a la personalización.

Desafortunadamente, no es posible crear un alias global. Puede definir alias en `/etc/profile`, pero no hay forma de hacer que formen parte del entorno para que sus definiciones se propaguen a los subprocesos del shell.

Sin embargo, puede configurar funciones globales. Estas son una excelente manera de personalizar el entorno de su sistema, porque las funciones son parte del shell, no procesos separados. Por ejemplo, es posible que desee hacer que *pushd* y *popd* (consulte el Capítulo 4 a través del Capítulo 6) estén disponibles de manera global.

La mejor manera de crear funciones globales es utilizar la variable integrada `FPATH` para la carga automática de funciones que presentamos en el Capítulo 4. Simplemente defina `FPATH` como un directorio de biblioteca de funciones, tal vez `/usr/local/functions`, y conviértalo en una variable de entorno *exportándola*. Luego, asegúrese de que el directorio enumerado en `FPATH` también esté incluido en `PATH`. En otras palabras, coloque este código o similar en `/etc/profile`:

```
FPATH=/usr/local/functions
PATH=$PATH:$FPATH
export FPATH PATH
```

Luego, coloque la definición de cada función global en un archivo en ese directorio con el mismo nombre que la función.

En cualquier caso, sugerimos utilizar funciones globales para la personalización global en lugar de scripts de shell. Dada la abundancia de memoria en la actualidad, no hay razón para no incluir funciones generalmente útiles como parte del entorno de sus usuarios.

10.3. Personalización de los modos de edición

Como vimos en el Capítulo 2, tienes la opción de utilizar los modos de edición de emacs o vi al editar tu línea de comandos. Además de los comandos disponibles en cada modo, puedes personalizar el comportamiento de los modos de edición según tus necesidades o entorno.

El Apéndice A aborda varios shells de terceros basados en el diseño de las shells Bourne y Korn. Esos shells generalmente proporcionan edición de línea de comandos, así como la capacidad de personalizar el editor mediante un comando integrado especial, un archivo de inicio especial o ambos.

El enfoque del shell de Korn es diferente. Se basa en un paradigma en el que programas el comportamiento que deseas del shell. Esto se logra mediante una trampa falsa, llamada KEYBD. Si existe, la trampa establecida para KEYBD se evalúa cuando *ksh* procesa caracteres normales de entrada de línea de comandos. [5] Dentro del código ejecutado para la trampa, dos variables especiales contienen el texto de la línea de comandos y el texto que se está ingresando y que causó la trampa. Variables adicionales especiales te permiten distinguir entre los modos *emacs* y *vi* e indicar la posición actual en la línea de entrada. Estas variables se enumeran en la Tabla 10.2.

Tabla 10.2: Variables de edición especiales

Variable	**Significado**
.sh.edchar	El carácter o secuencia de escape introducido por el usuario que causó la trampa KEYBD. El valor de `.sh.edchar` al final de la trampa se utiliza entonces para dirigir las acciones del editor incorporado.
.sh.edcol	La posición del cursor en la línea de entrada actual.
.sh.edmode	Igual a ESC en modo *vi*, vacío en caso contrario. (Utilice `[[ -n ${.sh.edmode} ]]` para probarlo).
.sh.edext	El texto de la línea de entrada actual.

[5] Los caracteres para cadenas de búsqueda y argumentos numéricos para comandos de modos *vi* y *emacs* no activan la trampa KEYBD.

Al ingresar en la trampa KEYBD, el contenido de `.sh.edchar` será un solo carácter, ESC seguido de un solo carácter o ESC, [, y un solo carácter. Puedes asignar un nuevo valor a `.sh.edchar` para cambiar la entrada que recibe el modo de edición actual. Así, la trampa KEYBD te permite interponer un «filtro» entre lo que ingresa el usuario y lo que los modos de edición del shell procesan realmente. El siguiente ejemplo es de la página 98 de «The New KornShell Command and Programming Language». [6] Presenta una función *keybind* que te permite asignar nuevas acciones a secuencias clave de entrada, similar al comando integrado *bind* de muchas otras shells.

```
# Citado de la Página 98 de
# "The New KornShell Command and Programming Language"

1 typeset -A Keytable
2 trap 'eval "${Keytable[${.sh.edchar}]}"' KEYBD
3 function keybind # key [action]
4  {
5    typeset key=$(print -f "%q" "$2")
6    case $# in
7    2)     Keytable[$1]=' .sh.edchar=${.sh.edmode}'"$key"
8           ;;
9    1)     unset Keytable[$1]
10          ;;
11   *)     print -u2 "Usage: $0 key [action]"
12          return 2 # los errores de uso devuelven 2 por defecto
13          ;;
14   esac
15 }
```

Esta es una función interesante. Vamos a revisarla línea por línea. La línea 1 crea un array asociativo para actuar como una tabla de pares clave/acción. La línea 2 establece la trampa KEYBD. Obtiene la acción del array asociativo y luego la ejecuta usando *eval*. La línea 3 inicia la función *keybind*, que toma uno o dos argumentos. Con dos argumentos, el segundo argumento se cita adecuadamente primero (línea 5, la variable *key* habría sido mejor nombrada `action`). La línea 7 luego crea la entrada en el array, utilizando `$1` (la secuencia de teclas del usuario) como índice y action citada como el valor a asignar a `.sh.edchar`. Observa cómo también se incluye `${.sh.mode}`. Esto tiene el efecto de forzar un cambio al modo de comando para el modo de edición *vi*. Es esta declaración de asignación generada la que se *evalúa* cada vez que se ejecuta la trampa.

El resto de la función es principalmente tareas administrativas: con un argumento (línea 9), se elimina la entrada dada en el array `Keytable`. Si hay más de dos argumentos (línea

[6] Este es el libro sobre *ksh93* escrito por David Korn y Morris Bolsky y publicado por Prentice Hall.

11), *keybind* imprime un mensaje y luego devuelve el valor (falso) 2.

Aunque algo inusual, el mecanismo de trampa KEYBD para tratar la entrada del usuario es tanto general como extensible; puedes hacer lo que quieras, simplemente como un Asunto Simple de Programación. Con otras shells, estás limitado a las facilidades integradas que proporcionan.

10.4. Funciones de seguridad del sistema

Características de Seguridad del Sistema La seguridad de Unix es un problema de notoria notoriedad. Prácticamente todos los aspectos de un sistema Unix tienen algún problema de seguridad asociado, y generalmente es trabajo del administrador del sistema preocuparse por este problema.

NOTA: Este no es un libro de texto sobre seguridad del sistema Unix. Ten en cuenta que esta sección apenas toca la punta del iceberg y que hay innumerables otros aspectos de la seguridad del sistema Unix además de cómo está configurada el shell. Consulta al final del capítulo un libro que recomendamos.

Primero presentamos una lista de «consejos» para escribir scripts de shell que tengan una mejor probabilidad de evitar problemas de seguridad. Luego cubrimos la *shell restringida*, que intenta poner un chaleco de fuerza alrededor del entorno del usuario. Luego presentamos la idea de un «caballo de Troya» y por qué se deben evitar tales cosas. Finalmente, discutimos el *modo privilegiado* del shell de Korn, que se utiliza con scripts de shell que se ejecutan como si el usuario fuera `root`.

10.4.1. Consejos para Scripts de Shell Seguros

Aquí tienes algunos consejos para escribir scripts de shell más seguros, cortesía del Profesor Eugene (Gene) Spafford, director del Centro de Educación e Investigación en Aseguramiento de la Información y Seguridad de la Universidad de Purdue [7]:

No incluyas el punto (*dot*)en PATH Este problema se describió en el Capítulo 3. Esto abre la puerta de par en par a «caballos de Troya», que se describen en la siguiente sección.

Protege los directorios `bin` Asegúrate de que cada directorio en `$PATH` solo sea escri-

[7] Ver http://www.cerias.purdue.edu.

bible por su propietario y por nadie más. Lo mismo se aplica a todos los programas *en* los directorios *bin*.

Diseña antes de codificar Dedica tiempo a pensar en lo que quieres hacer y cómo hacerlo; no simplemente escribas cosas con un editor de texto y sigas pirateando hasta que parezca funcionar. Incluye código para manejar errores y fallas de manera elegante.

Verifica la validez de todos los argumentos de entrada Si esperas un número, verifica que obtuviste un número. Comprueba que el número esté en el rango correcto. Haz lo mismo para otros tipos de datos; las instalaciones de expresiones regulares del shell son particularmente útiles para esto.

Verifica los códigos de error de todos los comandos que pueden devolver errores Cosas que no esperas que fallen podrían forzarse maliciosamente a fallar para hacer que el script se comporte de manera incorrecta. Por ejemplo, es posible hacer que algunos comandos fallen incluso como root si el argumento es un disco montado por NFS o un archivo de dispositivo orientado a caracteres.

No confíes en las variables de entorno proporcionadas Verifícalas y restablécelas a valores conocidos si son utilizadas por comandos posteriores (por ejemplo, `TZ, FPATH, PATH, IFS`, etc.). El shell Korn restablece automáticamente `IFS` a su valor predeterminado al inicio, ignorando lo que estaba en el entorno, pero muchas otras shells no lo hacen. En todos los casos, es una excelente idea establecer explícitamente `PATH` para que contenga solo los directorios *bin* del sistema.

Comienza en un lugar conocido Cambia explícitamente al directorio conocido cuando el script comienza para que cualquier ruta relativa posterior sea a un lugar conocido. Asegúrate de que el *cd* tenga éxito:

```
cd app-dir || exit 1
```

Usa rutas completas para los comandos Haz esto para saber qué versión estás obteniendo, independientemente de `$PATH`.

Utiliza *syslog(8)* para mantener un registro de auditoría Registra la fecha y hora de la invocación, el nombre de usuario, etc.; consulta *logger(1)*. Si no tienes *syslog*, crea una función para mantener un archivo de registro:

```
function logsys {
  print -r -- "$@" >> /var/adm/logsysfile
}
```

```
logsys "Ejecutado por el usuario " $(/bin/whoami) "($USER) en " $(/bin/date)
```

(*whoami(1)* imprime el nombre de inicio del usuario efectivo, un concepto descrito más adelante en este capítulo).

Siempre entrecomilla la entrada del usuario al usar esa entrada Por ejemplo, `"$1"` y `"$*"`. Esto evita que la entrada del usuario maliciosa se evalúe y ejecute aún más.

No uses `eval` en la entrada del usuario Además de entrecomillar la entrada del usuario, no la pases a el shell para volver a procesarla con *eval*. Si el usuario lee tu script y ve que usa *eval*, es fácil subvertir el script para que haga casi cualquier cosa.

Entrecomilla los resultados de la expansión de comodines Hay varias cosas desagradables que puedes hacer a un administrador del sistema creando archivos con espacios, puntos y comas, comillas invertidas, etc., en los nombres de archivos. Si los scripts administrativos no entrecomillan los argumentos de los nombres de archivo, los scripts pueden dañar o revelar el sistema.

Verifica la entrada del usuario en busca de metacaracteres Busca metacaracteres como `$` o `` ` `` (sustitución de comandos antigua) si utilizas la entrada en un *eval* o `$(...)`.

Prueba tu código y léelo críticamente Busca suposiciones y errores que puedan ser explotados. Ponte de mal humor y lee tu código con la intención de tratar de descubrir cómo subvertirlo. Luego corrige cualquier problema que encuentres.

Ten en cuenta las condiciones de carrera Si un atacante puede ejecutar comandos arbitrarios entre dos comandos en tu script, ¿comprometerá la seguridad? Si es así, busca otra forma de hacerlo.

Desconfía de los enlaces simbólicos Cuando *cambias* los permisos o editas un archivo, verifícalo para asegurarte de que sea un archivo y no un enlace simbólico a un archivo crítico del sistema. (Utiliza `[[ -L file ]]` o `[[ -h file ]]` para comprobar si *file* es un enlace simbólico).

Haz que otra persona revise tu código en busca de errores A menudo, un par de ojos frescos puede detectar cosas que el autor original de un programa pasa por alto.

Usa *setgid* en lugar de *setuid*, si es posible Estos términos se discuten más adelante en este capítulo. En resumen, al usar *setgid*, restringes la cantidad de daño que se puede hacer al grupo que está comprometido.

Utiliza un usuario nuevo en lugar de `root` Si debes usar *setuid* para acceder a un grupo de archivos, considera crear un usuario nuevo, *no root*, con ese propósito, y establece *setuid* en él.

Limita el código *setuid* tanto como sea posible Reduce al mínimo la cantidad de código *setuid* que puedas. Muévelo a un programa separado e invócalo desde un script más grande cuando sea necesario. Sin embargo, ¡asegúrate de programar defensivamente como si el script pudiera ser invocado por cualquiera desde cualquier lugar!

Chet Ramey, el mantenedor de *bash*, ofrece el siguiente prólogo para usar en scripts de shell que necesitan ser más seguros:

```
# restablecer IFS, aunque ksh no importa IFS del entorno,
# $ENV podría establecerlo
IFS=$' \t\n'

# asegurarse de que unalias no sea una función, ya que es un comando incorporado regular
# unset es un comando incorporado especial, por lo que se encontrará antes que las funciones
unset -f unalias

# deshacer todos los alias
# citar unalias para que no sea alias-expandido
\unalias -a

# asegurarse de que command no sea una función, ya que es un comando incorporado regular
# unset es un comando incorporado especial, por lo que se encontrará antes que las funciones
unset -f command

# obtener un prefijo de ruta confiable, manejando el caso en que getconf no está disponible
# (no es demasiado necesario, ya que getconf es un comando incorporado de ksh93)
SYSPATH="$(command -p getconf PATH 2>/dev/null)"
if [[ -z "$SYSPATH" ]]; then
  SYSPATH="/usr/bin:/bin"       # elige tu veneno
fi
PATH="$SYSPATH:$PATH"
```

Shell restringida

El shell restringida está diseñada para colocar al usuario en un entorno donde su capacidad para moverse y escribir archivos esté severamente limitada. Generalmente se utiliza para cuentas de invitados. Cuando se invoca como *rksh* (o con la opción `-r`), *ksh* actúa como una shell restringida. Puedes hacer que el shell de inicio de sesión de un usuario sea restringida colocando la ruta completa de *rksh* en la entrada `/etc/passwd` del usuario. El archivo ejecutable *ksh* debe tener un enlace a él con el nombre *rksh* para que esto funcione.

Las restricciones específicas impuestas por el shell restringida impiden al usuario hacer lo siguiente:

- Cambiar directorios de trabajo: *cd* no está operativo. Si intentas usarlo, obtendrás el mensaje de error `ksh: cd: restricted`.
- Redirigir la salida a un archivo: los redireccionadores `>, >|, <>, y >>` no están permitidos. Esto incluye el uso de *exec*.
- Asignar un nuevo valor a las variables de entorno `ENV, FPATH, PATH` o `SHELL`, o intentar cambiar sus atributos con *typeset*.
- Especificar rutas de comandos con barras inclinadas (/) en ellas. El shell solo ejecuta comandos encontrados a lo largo de `$PATH`.
- Agregar nuevos comandos incorporados con el comando *builtin*. (Esta función muy avanzada está fuera del alcance de este libro).

Estas restricciones entran en vigencia después de que se ejecutan los archivos *.profile* y de entorno del usuario. Esto significa que el entorno completo del usuario del shell restringida se configura en *.profile*. Esto permite que el administrador del sistema configure el entorno según considere conveniente.

Para evitar que el usuario sobrescriba `~/.profile`, no es suficiente hacer que el archivo sea de solo lectura para el usuario. O bien, el directorio principal no debe ser escribible por el usuario, o los comandos en `~/.profile` deben *cd* a un directorio diferente.

Dos formas comunes de configurar dichos entornos son configurar un directorio de comandos «seguros» y hacer que ese directorio sea el único en `PATH`, y configurar un menú de comandos del cual el usuario no pueda escapar sin salir del shell. En cualquier caso, asegúrate de que no haya otra shell en ningún directorio enumerado en `$PATH`; de lo contrario, el usuario puede simplemente ejecutar esa shell y evitar las restricciones mencionadas anteriormente.

ADVERTENCIA: Aunque la capacidad de restringir el shell ha estado disponible (si no necesariamente compilada o documentada) desde la versión original 7 de Bourne shell, rara vez se usa. Configurar un entorno utilizable pero correctamente restringido es difícil en la práctica. Entonces, *caveat emptor*.

10.4.2. Caballos de Troya

El concepto de un *caballo de Troya* fue introducido brevemente en el Capítulo 3. Un caballo de Troya es algo que parece inofensivo, o incluso útil, pero que contiene un peligro oculto.

Considera el siguiente escenario. El usuario John Q. Programmer (nombre de inicio de sesión `jprog`) es un excelente programador y tiene una gran colección de programas personales en `~jprog/bin`. Este directorio ocurre primero en la variable `PATH` en `~jprog/.profile`. Dado que es un buen programador, la administración lo ascendió recientemente a administrador del sistema.

Este es un campo completamente nuevo de actividad, y John, sin saberlo mejor, lamentablemente dejó su directorio *bin* escribible. Llega W. M. Badguy, quien crea el siguiente script de shell, llamado *grep*, en el directorio *bin* de John:

```
/bin/grep "$@"
case $(whoami) in                  # Verifica el nombre del ID de usuario efectivo
root)    Cosas malas aquí          # ¡Peligro, Will Robinson, peligro!
         rm ~/jprog/bin/grep       # Ocultar la evidencia
         ;;
esac
```

En sí mismo, este script no puede causar daño cuando *jprog* está trabajando como *él mismo*. El problema surge cuando *jprog* usa el comando *su(1)*. Este comando permite a un usuario regular «cambiar de usuario» a un usuario diferente. Por defecto, permite a un usuario regular convertirse en `root` (siempre y cuando ese usuario conozca la contraseña, por supuesto). El problema es que normalmente, *su* usa cualquier `PATH` que hereda.[8] En este caso, `$PATH` incluye `~jprog/bin`. Ahora, cuando `jprog`, trabajando como `root`, ejecuta *grep*, en realidad ejecuta la versión del *caballo de Troya* en su *bin*. Esta versión ejecuta el *grep* real, por lo que `jprog` obtiene los resultados que espera. Pero también ejecuta silenciosamente la parte de cosas malas aquí, como `root`. Esto significa que Unix permitirá que el script haga lo que quiera. *Cualquier cosa*. Y para empeorar las cosas, al eliminar el caballo de Troya cuando ha terminado, ya no hay evidencia.

Los directorios *bin* escribibles abren una puerta para los *caballos de Troya*, al igual que tener un punto en el `PATH`. Tener scripts de shell escribibles en cualquier directorio bin es otra puerta. Así como cierras y aseguras las puertas de tu casa por la noche, ¡asegúrate de cerrar cualquier puerta en tu sistema!

[8] Adquiere el hábito de usar `su - user` para cambiar al usuario como si el usuario estuviera haciendo un inicio de sesión real. Esto evita la importación del `PATH` existente.

Setuid y modo privilegiado

Muchos problemas de seguridad de Unix se centran en un atributo de archivo de Unix llamado el bit *setuid* (set user ID). Es como un bit de permiso (consulta la discusión anterior sobre *umask*): cuando un archivo ejecutable lo tiene activado, el archivo se ejecuta con un ID de usuario efectivo igual al propietario del archivo. El ID de usuario efectivo es distinto del ID de usuario real del proceso, y Unix aplica sus pruebas de permisos al ID de usuario efectivo del proceso.

Por ejemplo, supongamos que has escrito un programa de juego muy ingenioso que mantiene un archivo de puntuación privado mostrando a los 15 mejores jugadores en tu sistema. No quieres que el archivo de puntuación sea escribible por cualquier persona, porque cualquiera podría llegar y editarlo para convertirse en el jugador principal. Haciendo tu juego setuid a tu ID de usuario, el programa del juego puede actualizar el archivo, que es de tu propiedad, pero nadie más puede actualizarlo. (El programa del juego puede determinar quién lo ejecutó al mirar su ID de usuario real y usar eso para determinar el nombre de inicio de sesión).

La facilidad *setuid* es una característica agradable para juegos y archivos de puntuación, pero se vuelve mucho más peligrosa cuando se usa para `root`. Hacer programas *setuid* `root` permite a los administradores escribir programas que realizan ciertas tareas que requieren privilegios de `root` (por ejemplo, configurar impresoras) de manera controlada. Para establecer el bit *setuid* de un archivo, el superusuario puede escribir `chmod 4755 nombrearchivo`; el 4 es el bit setuid.

Existe una facilidad similar a nivel de grupo, conocida (no sorprendentemente) como *setgid* (set group ID). Usa `chmod 2755 nombrearchivo` para activar los permisos *setgid*. Cuando haces un `ls -l` en un archivo *setuid* o *setgid*, la *x* en el modo de permisos se reemplaza con una *s*; por ejemplo, `-rws--s--x` para un archivo que es legible y escribible por el propietario, ejecutable por todos y tiene ambos bits *setuid* y *setgid* activados (modo octal 6711).

La sabiduría moderna de la administración del sistema dice que crear scripts de shell *setuid* y *setgid* es una muy, muy mala idea. Esto ha sido especialmente cierto bajo la C shell, porque su archivo de entorno *.cshrc* introduce numerosas oportunidades para intrusiones. En particular, hay varias formas de engañar a un script de shell *setuid* para que se convierta en una shell *interactiva* con un ID de usuario efectivo de `root`. Esto es lo mejor que un

cracker podría esperar: la capacidad de ejecutar cualquier comando como `root`.

NOTA: Existe una diferencia importante entre un script de shell *setuid* y una shell *setuid*. Esta última es una copia del ejecutable del shell, que se ha hecho pertenecer a `root` y se le ha aplicado el bit *setuid*. En la sección anterior sobre caballos de Troya, supongamos que la parte de *cosas malas* aquí era este código:

```
cp /bin/ksh ~badguy/bin/myls
chown root ~badguy/bin/myls
chmod 4755 ~badguy/bin/myls
```

Recuerda, este código se ejecuta como `root`, por lo que funcionará. Cuando `badguy` ejecuta *myls*, es un archivo ejecutable en código de máquina, y se honra el bit setuid. Hola shell que se ejecuta como `root`. ¡Adiós seguridad!

El *modo privilegiado* fue diseñado para proteger contra scripts de shell *setuid*. Esta es una opción *set -o* (`set -o privileged` o `set -p`), pero el shell la activa automáticamente cada vez que ejecuta un script cuyo bit *setuid* está activado, es decir, cuando el ID de usuario efectivo es diferente del ID de usuario real.

En el *modo privilegiado*, cuando se invoca un script de shell Korn *setuid*, el shell ejecuta el archivo `/etc/suid_profile`. Este archivo debe escribirse de manera que restrinja los scripts de shell *setuid* de manera similar a como lo hace la shell restringida. Como mínimo, debería hacer que `PATH` sea de solo lectura (`typeset -r PATH` o `readonly PATH`) y configurarlo en uno o más directorios «seguros». Una vez más, esto evita que se invoquen señuelos.

Dado que el modo privilegiado es una opción, es posible desactivarlo con el comando `set +o privileged` (o `set +p`). Pero esto no ayuda al posible cracker del sistema: el shell cambia automáticamente su ID de usuario efectivo para que sea igual al ID de usuario real, es decir, si desactivas el modo privilegiado, también desactivas *setuid*.

Además del modo privilegiado, *ksh* proporciona un programa «agente» especial, `/etc/suid_exec`, que ejecuta scripts de shell *setuid* (o scripts de shell que son ejecutables pero no legibles).

Para que esto funcione, el script no debe comenzar con `#! /bin/ksh`. Cuando se invoca el programa, *ksh* intenta ejecutar el programa como un ejecutable binario regular. Cuando el sistema operativo no puede ejecutar el script (porque no es binario y porque no tiene el nombre de un intérprete especificado con `#!`), *ksh* se da cuenta de que es un script e invoca `/etc/suid_exec` con el nombre del script y sus argumentos. También se encarga de pasar un «token» de autenticación a `/etc/suid_exec` que indica los ID de usuario y grupo reales

y efectivos del script. `/etc/suid_exec` verifica que es seguro ejecutar el script y luego se encarga de invocar *ksh* con los ID de usuario y grupo reales y efectivos adecuados en el script.

Aunque la combinación de modo privilegiado y `/etc/suid_exec` te permite evitar muchos de los ataques a scripts *setuid*, escribir scripts que se puedan ejecutar *setuid* de manera segura es un arte difícil, que requiere una buena cantidad de conocimiento y experiencia. Debería hacerse con mucho cuidado.

De hecho, los peligros de scripts de shell *setuid* y *setgid* (al menos para shells además de *ksh*) son tan grandes que los sistemas Unix modernos, tanto los sistemas Unix comerciales como los clones de software libre (derivados de 4.4-BSD y GNU/Linux), desactivan los bits *setuid* y *setgid* en scripts de shell. Incluso si aplicas los bits al archivo, el sistema operativo no los respeta. [9]

Aunque los scripts de shell *setuid* no funcionan en sistemas modernos, hay ocasiones en las que el modo privilegiado sigue siendo útil. En particular, existe un programa de terceros ampliamente utilizado llamado *sudo*, que, citando la página web, «permite a un administrador del sistema dar a ciertos usuarios (o grupos de usuarios) la capacidad de ejecutar algunos (o todos) los comandos como `root` u otro usuario mientras registra los comandos y argumentos». La página de inicio de *sudo* es http://www.courtesan.com/sudo/. Un administrador del sistema podría ejecutar fácilmente `sudo /bin/ksh -p` para obtener un entorno conocido para realizar tareas administrativas.

Finalmente, si deseas aprender más sobre la seguridad de Unix, recomendamos *Practical UNIX & Internet Security* de Simson Garfinkel y Gene Spafford. Está publicado por O'Reilly & Associates.

[9] MacOS X parece ser una notable excepción. ¡Ten mucho cuidado si ejecutas uno o más de esos sistemas!

CAPÍTULO A

SHELLS RELACIONADOS

La fragmentación del mercado Unix ha tenido sus ventajas y desventajas. Las ventajas vinieron principalmente en los primeros días: la falta de estandarización y la proliferación entre académicos y profesionales técnicamente competentes contribuyeron a un «mercado libre» saludable para el software de Unix, en el que varios programas del mismo tipo (por ejemplo, shells, editores de texto, herramientas de administración del sistema) a menudo competían por popularidad. Los mejores programas generalmente se volvían más ampliamente utilizados, mientras que el software inferior tendía a desaparecer.

Pero a menudo no había un único programa «mejor» en una categoría dada, por lo que varios prevalecían. Esto llevó a la situación actual, donde la multiplicidad de software similar ha llevado a la confusión, la falta de compatibilidad y, lo más desafortunado de todo, a la incapacidad de Unix para capturar una parte tan grande del mercado como otras plataformas operativas. En particular, Unix se ha relegado a su posición actual como un sistema operativo muy popular para servidores, pero es una rareza en las computadoras de escritorio.

La categoría «shell» probablemente ha sufrido de esta manera más que cualquier otro tipo de software. Como dijimos en el Prefacio y el Capítulo 1, una de las fortalezas de Unix es que el shell es reemplazable y, por lo tanto, actualmente hay varios shells disponibles; las diferencias entre ellos a menudo no son tan grandes. Creemos que el shell de Korn es uno de los mejores entre los shells más utilizados, pero otros shells ciertamente tienen sus seguidores leales, por lo que no es probable que caigan en el olvido. De hecho, parece que los shells, compatibles con Bourne o no, continúan proliferando.

Por lo tanto, sentimos que era necesario incluir información sobre shells similares al shell de Korn. Este apéndice resume las diferencias entre el shell de Korn y los siguientes shells:

- El Shell de Bourne de System V Release 4, como una especie de referencia.
- La versión de 1988 del shell de Korn.
- El Estándar del Shell IEEE POSIX 1003.2, al que se adhieren el shell de Korn y otros shells.
- El Shell Desktop Korn (*dtksh*), un shell de Korn con mejoras para programación en el sistema X Window, como parte del Common Desktop Environment (CDE).
- El shell *tksh*, una interesante combinación de *ksh93* con Tcl/Tk.
- *pdksh*, una versión de dominio público ampliamente utilizada del shell de Korn.
- El shell *bash*, otro shell mejorado de Bourne con algunas características del shell C y Korn.
- El shell Z, *zsh*, otro shell mejorado de Bourne con algunas características de los shells C y Korn y muchas, muchas más características propias en plataformas de PC de escritorio.
- Similares al shell de Korn en plataformas de PC de sobremesa

A.1. El Shell de Bourne

El shell de Korn es casi completamente compatible hacia atrás con el Shell de Bourne. La única característica significativa de este último que el shell de Korn no admite es ^ (acento circunflejo) como sinónimo del carácter pipe (|).[1] Esta es una característica arcaica que el Shell de Bourne incluye por su propia compatibilidad hacia atrás con shells anteriores. Ninguna versión moderna de Unix tiene código de shell que use ^ como pipe.

Para describir las diferencias entre el Shell de Bourne y el shell de Korn, revisaremos cada capítulo de este libro y enumeraremos las características discutidas en el capítulo que el Shell de Bourne no admite.

1. Capítulo 1
    - Las formas `cd old new` y `cd -` del comando *cd*.
    - Expansión de tilde (~).

[1] También hay algunas diferencias en cómo reaccionan los dos shells ante ciertas entradas extremadamente patológicas. Por lo general, el shell de Korn procesa correctamente lo que hace que el Shell de Bourne «se atragante».

- El Shell de Bourne siempre sigue el diseño físico de archivos, lo que afecta lo que sucede cuando haces `cd ..` fuera de algún lugar que era un enlace simbólico.
- Los comandos incorporados no tienen ayuda en línea.
- Algunas versiones antiguas del Shell de Bourne no admiten el comando *jobs* y el control de trabajos, o pueden requerir ser invocadas como *jsh* para habilitar las características de control de trabajos.

2. Capítulo 2
   - Todos (es decir, el Shell de Bourne no admite ninguna de las características de historial y edición discutidas en el Capítulo 2).
3. Capítulo 3
   a) No se admiten alias.
   b) Las opciones `set -o` no funcionan. El Shell de Bourne admite las abreviaturas enumeradas en la tabla de «Opciones» en el Apéndice B, excepto *-A, -b, -C, -m, -p* y *-s*.
   c) No se admiten archivos de entorno; tampoco el comando *print* (usa echo en su lugar). Las siguientes variables internas no se admiten:

.sh.edchar	.sh.version	HISTEDIT	LINENO	PS4
.sh.edcol	COLUMNS	HISTFILE	LINES	PWD
.sh.edmode	EDITOR	HISTSIZE	OLDPWD	PWD
.sh.edtext	ENV	LANG	OPTARG	RANDOM
.sh.match	FCEDIT	LC_ALL	OPTIND	SECONDS
.sh.name	FIGNORE	LC_COLLATE	PPID	TMOUT
.sh.subscript	FPATH	LC_CTYPE	PS3	VISUAL
.sh.value	HISTCMD	LC_NUMERIC		

   d) Algunas de estas variables (por ejemplo, `EDITOR` y `VISUAL`) aún son utilizadas por otros programas, como *mail* y noticias.
4. Capítulo 4
   - No están disponibles los nombres de variables extendidas (aquellas con un punto en ellas), así como la asignación de variables compuestas y la concatenación de cadenas con el operador `+=`.
   - No se admiten variables indirectas (referencias de nombres).

- No está disponible el comando *whence*; usa *type* en su lugar.
- Los operadores variables de coincidencia de patrones (%, %, #, ##, etc.) y los caracteres comodín avanzados (expresión regular) no están disponibles; usa el comando externo *expr* en su lugar.
- No se admiten funciones cargadas automáticamente, y solo se pueden usar funciones de estilo POSIX (definidas utilizando la sintaxis *name()*).
- La sintaxis de sustitución de comandos es diferente: utilice el antiguo `` `command` `` en lugar de `$(command)`. (Algunos proveedores han mejorado sus shells Bourne para que admitan la notación `$(comando)`, ya que está definida por el estándar POSIX).

5. Capítulo 5

- `return` solo se puede usar desde dentro de una función.
- Las pruebas condicionales utilizan la sintaxis antigua: `[ condición ]` o `test condición` en lugar de `[[ condición ]]`. Estas son realmente dos formas del mismo comando (consulta la página de manual de *test(1)*).
- Los operadores lógicos `&&` y `||` son `-a` y `-o` en su lugar. Los operadores de prueba admitidos difieren de un sistema a otro.[2]
- La palabra clave ! para invertir el estado de salida de un comando no estaba en el Shell de Bourne de SVR4, aunque puede estar disponible en tu sistema, ya que es requerida por POSIX.
- No se admite la construcción `select`. Tampoco el bucle `for` aritmético, y no hay forma de pasar de un caso a otro dentro de una declaración `case`.
- No hay un equivalente para `TMOUT`.

6. Capítulo 6

- El comando *getopts* del Shell de Bourne de SVR4 es similar, pero no idéntico, al de *ksh*. No permite opciones que comienzan con un signo más, ni ninguna de las características avanzadas descritas en el Apéndice B.

[2]En el Bourne Shell original de la Versión 7 (y en sistemas Unix de Berkeley hasta 4.3BSD), *test* y `[ condición ]` eran en realidad comandos externos. (Eran enlaces duros entre sí en */bin*.) Sin embargo, se integraron en el Shell de Bourne en todos los sistemas desde System III (circa 1981).

- No se admite la aritmética: usa el comando externo *expr* en lugar de la sintaxis `$((...))`. Para condicionales numéricas, usa la antigua sintaxis de prueba de condición y los operadores relacionales `-lt, -eq`, etc., en lugar de `((...))`. No se admite *let*.
- No se admiten variables de matriz y el comando *typeset*.

7. Capítulo 7

- No se admiten los siguientes redireccionadores de E/S[3]:

`>|`	`<>`	`<<<`	`<&p`	`>&p`	`<&n-`	`>&n-`	`|&`

- No se admite *print* (usa *echo* en su lugar). *printf* generalmente está disponible como un comando externo.
- Ninguna de las opciones de *read* es compatible, ni la capacidad de proporcionar un indicador con el nombre de la primera variable.
- El Shell de Bourne no hace interpretaciones especiales de rutas, como `/dev/fd/2` o `/dev/tcp/ftp.oreilly.com/ftp`.
- Los mecanismos de comillas `$"..."` y `$'...'` no están disponibles.

8. Capítulo 8

- Se admite el control de trabajos, pero solo si el shell se invoca con la opción `-j` o como *jsh*.
- La opción `-` a *trap* (restablecer *trap* al valor predeterminado para esa señal) no está disponible. En su lugar, la falta de un *trap* indica que se deben restablecer las trampas suministradas. *trap* acepta solo números de señales, no nombres lógicos.
- No se admiten las corutinas.
- El comando *wait* solo acepta identificadores de procesos.

9. Capítulo 9

[3]El operador `<>` estaba en el Shell de Bourne original de la Versión 7, pero no estaba documentado, ya que no siempre funcionaba correctamente en todos los sistemas Unix. No se debe depender de su disponibilidad para la programación de Shell de Bourne.

- Las señales ERR y DEBUG no están disponibles. La señal EXIT es compatible, como la señal 0.
- `set -n` tiene efecto incluso en shells interactivos.
- La salida de la traza con `set -x` no se puede personalizar (no hay variable `PS4`).
- Las funciones de disciplina no están disponibles.

10. Capítulo 10
    - No se admite el modo privilegiado (algunos shells Bourne lo tienen).
    - Los comandos *ulimit* y *umask* no son tan capaces.
    - No está disponible la trampa `KEYBD`.

A.2. El Shell Korn de 1988

Quizás el shell más evidente con el que comparar *ksh93* sea *ksh88*, la versión de 1988 del shell de Korn. Esta sección describe brevemente aquellas características de *ksh93* que son diferentes o inexistentes en *ksh88*. Al igual que con la presentación para el shell de Bourne, los temas se cubren en el mismo orden en que se presentan en el resto del libro.

1. Capítulo 1
    - Las facilidades de ayuda integradas (como *-?*, *–man*, y así sucesivamente) no están disponibles.
    - La sustitución de tilde no ocurre durante la expansión de variables (`${var op word}`).
2. Capítulo 2
    - CTRL-T funciona de manera diferente en el modo de edición emacs. En *ksh88*, transpone los dos caracteres a la derecha del punto e mueve el punto hacia adelante en un carácter.
    - En el modo emacs, ESC ESC, ESC *, y ESC = no funcionan en la primera palabra de una línea de comando, es decir, no hay una facilidad de autocompletado de comandos. El comando ESC # siempre antepone un carácter #. Nunca los elimina.

- De manera similar, en el modo *vi*, los comandos *, \ y = no funcionan en la primera palabra de una línea de comando, y el comando # siempre antepone un carácter #. Nunca los elimina.
- Finalmente, no hay autocompletado de variables, el comando *hist* se llama *fc*, `FCEDIT` se usa en lugar de `HISTEDIT`, y la variable `HISTCMD` no está disponible.

3. Capítulo 3

- El archivo `ENV` se carga para todos los shells, y *ksh88* solo realiza la sustitución de variables en el valor de `$ENV`.
- El seguimiento de alias se puede desactivar en *ksh88*.
- La opción `-p` para alias no está presente en *ksh88*, al igual que la opción `-a` para unalias.
- Las siguientes variables internas no son compatibles:

.sh.edchar	.sh.match	.sh.version	LANG
.sh.edcol	.sh.name	FIGNORE	LC_ALL
.sh.edmode	.sh.subscript	HISTCMD	LC_COLLATE
.sh.edtext	.sh.value	HISTEDIT	LC_CTYPE

4. Capítulo 4

- En *ksh88*, ambas sintaxis para definir funciones producen funciones con semántica de shell de Korn. No puedes aplicar el comando dot a un nombre de función.
- El orden de búsqueda de comandos en *ksh88* es palabras clave, aliases, todos los comandos internos, funciones, y luego comandos externos y scripts. El orden fue cambiado en *ksh93* para cumplir con POSIX.
- En *ksh88*, las funciones no definidas (*autoloaded*) se buscan exclusivamente a lo largo de la lista en la variable `FPATH`, y `PATH` no está involucrado.
- La característica del archivo *.paths* no está disponible.
- Muchas de las características de sustitución de variables descritas en el texto principal son nuevas en *ksh93*. Solo las siguientes están disponibles en *ksh88*: `${name:-string}`, `${name:=string}`, `${name:?string}`, `${name:+string}`,

`${name#pattern}`, `${name##pattern}`, `${name%pattern}`, `${name%%pattern}`, `${#name}`, `${#name[*]}`, y `${#name[@]}`.

- Variables compuestas, namerefs y el operador += para agregar a una variable no están en *ksh88*.
- La notación \n en patrones no está disponible, al igual que la coincidencia no codiciosa, el uso de escapes de barra invertida en patrones, opciones en subpatrones, ni ninguna de las clases de caracteres [:...:].
- El array `.sh.match` no está disponible.

5. Capítulo 5
   - Los valores de salida en *ksh88* solo llegan hasta 128. Los programas que mueren debido a una señal tienen un estado de salida de 128 más el número de señal.
   - No hay un *comando* built-in. Para reemplazar un built-in, debes usar una combinación incómoda de alias, funciones y citas.
   - La opción `set -o pipefail` y la palabra clave ! para invertir el sentido de una prueba no están disponibles. En lugar del operador `==` para `[[...]]`, *ksh88* usa `=`.
   - El bucle `for` aritmético no está en *ksh88*. El uso de `;&` para pasar de un caso a otro existía en *ksh88* pero no estaba documentada. La variable `TMOUT` existía en *ksh88*, pero solo se aplicaba al shell en sí, no al bucle `select` o al comando `read`.
6. Capítulo 6
   - La versión de *getopts* de *ksh88* no tenía la capacidad de especificar argumentos numéricos para opciones, ni una forma de especificar argumentos opcionales para opciones.
   - La aritmética integrada solo admite enteros, y los operadores ++, –, ?: y coma no están disponibles. Las constantes numéricas de la forma `base#número` solo pueden llegar hasta la base 36. No hay funciones aritméticas integradas.
   - Solo existen arrays indexados, y el índice máximo es 1023.
7. Capítulo 7
   - No se admiten los siguientes redireccionadores de E/S:

- `<&n-`
- `>&n-`
- `<<<`

- La red de TCP/IP está disponible a partir de *ksh88e*, pero debes usar direcciones IP.
- *ksh88* no tiene el comando *printf*, ni las opciones `-A, -d, -n` y `-t` para *read*. `TMOUT` no afecta a *read*.
- La expansión de llaves y la sustitución de procesos son opciones de tiempo de compilación que generalmente no están disponibles en *ksh88*.
- La traducción de locales con `$"..."` y las cadenas ANSI C con `$'...'` no están disponibles.

8. Capítulo 8
   - El comando *disown* no está disponible, y tampoco las opciones `-n` y `-s` para *kill*. `kill -l` solo se puede usar por sí mismo para enumerar las señales disponibles.
   - *true* y *false* son alias predefinidos en lugar de comandos internos.
   - Las funciones y los alias se exportan a los subprocesos del shell; esto no es cierto en *ksh93*.
9. Capítulo 9
   - Las trampas para la señal falsa DEBUG se ejecutan *después* de que se ejecuta cada comando, no antes.
   - Las funciones de disciplina no están disponibles.
10. Capítulo 10
    - El comando *umask* solo funciona con máscaras octales.
    - No puedes personalizar los editores integrados; la señal falsa KEYBD no existe.

A.3. El Estándar del Shell POSIX IEEE 1003.2

Ha habido muchos intentos de estandarizar Unix. Los intentos monolíticos de dominación del mercado por parte de empresas de hardware, frágiles coaliciones industriales, fallas de

marketing y otros esfuerzos por el estilo son parte de la historia, y también de la frustración.

Solo un esfuerzo de estandarización no ha estado vinculado a intereses comerciales: la Interfaz Portátil del Sistema Operativo, conocida como POSIX. Este esfuerzo comenzó en 1981 con el Comité de Normas de `/usr/group` (ahora UniForum), que produjo la Norma de `/usr/group` tres años después. La lista de contribuyentes creció. Con el tiempo, el esfuerzo por crear un estándar formal se trasladó bajo el paraguas del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) y la Organización Internacional de Normalización (ISO).

El primer estándar POSIX se publicó en 1988 y se revisó en 1996. Este, llamado Estándar IEEE 1003.1, abordaba problemas a nivel de llamadas al sistema. El Estándar IEEE 1003.2, que cubre el shell, programas de utilidad y problemas de interfaz de usuario, fue ratificado en septiembre de 1992 después de seis años de esfuerzo. En septiembre de 2001, se aprobó una revisión conjunta de ambos estándares. El nuevo estándar, que cubre todo el material de los dos documentos anteriores, se conoce como Estándar IEEE 1003.1-2001.

Los estándares POSIX nunca se pensaron como rígidos y absolutos. Los miembros del comité ciertamente no iban a poner armas en la cabeza de los implementadores del sistema operativo y obligarlos a adherirse. En cambio, los estándares están diseñados para ser lo suficientemente flexibles como para permitir tanto la coexistencia de software similar disponible, para que el código existente no esté en peligro de volverse obsoleto, como la adición de nuevas características, para que los proveedores tengan incentivos para innovar. En otras palabras, se supone que son el tipo de estándares de terceros que los proveedores podrían estar interesados en seguir.

Como resultado, la mayoría de los proveedores de Unix cumplen actualmente con ambos estándares. El shell de Korn no es una excepción; se pretende que sea 100 % compatible con POSIX. Conviene estar familiarizado con lo que hay en el estándar si quieres escribir código que sea portable en diferentes sistemas.

La parte del shell del estándar describe utilidades que deben estar presentes en todos los sistemas y otras que son opcionales, según la naturaleza del sistema. Una de esas opciones es la opción de Utilidades de Portabilidad del Usuario, que define estándares para el uso interactivo del shell y utilidades interactivas como el editor vi. El estándar, de aproximadamente 2000 páginas, está disponible a través del IEEE; para obtener información, ponte en contacto con el IEEE:

IEEE Customer Service
445 Hoes Lane, PO Box 1331
Piscataway, NJ 08855-1331
(800) 678-IEEE (Estados Unidos y Canadá)
(732) 981-0060 (internacional/local)
(732) 981-9667 (fax)

customer.service@ieee.org

http://www.standards.ieee.org/catalog/ordering.html

Los miembros del comité tenían dos factores motivadores que considerar cuando diseñaron el estándar del shell. Por un lado, el diseño debía acomodar, tanto como fuera posible, el código del shell existente escrito bajo varios shells derivados de Bourne (Version 7, System V, BSD y Korn). Estos shells son diferentes de varias maneras extremadamente sutiles, la mayoría de las cuales tienen que ver con la forma en que ciertos elementos sintácticos interactúan entre sí.

Debe haber sido bastante difícil y tedioso especificar estas diferencias, sin mencionar llegar a compromisos entre ellas. Agrega los sesgos de algunos miembros del comité hacia shells particulares, y podrías entender por qué llevó seis años ratificar el primer estándar 1003.2 y otros cinco años para fusionar los dos estándares.

Por otro lado, el diseño del shell debía servir como estándar en el que basar futuras implementaciones de shell. Esto implicaba objetivos de simplicidad, claridad y precisión, objetivos que parecen especialmente esquivos en el contexto de los problemas mencionados anteriormente.

Los diseñadores encontraron una manera de aliviar este dilema: decidieron que el estándar debería incluir no solo las características incluidas en el shell, sino también aquellas explícitamente omitidas y aquellas incluidas pero con funcionalidad no especificada. La última categoría permite que algunas innovaciones de shells existentes «se cuelen» sin formar parte del estándar, mientras que listar las características omitidas ayuda a los programadores a determinar qué características en los scripts de shell existentes no serán portables a los shells futuros.

El estándar POSIX se basa principalmente en el shell de Bourne de System V. Por lo tanto, debes asumir que las características del shell de Korn que no están presentes en el shell de Bourne tampoco están incluidas en el estándar POSIX.

Sin embargo, el shell de Korn contribuyó con algunas de sus características al estándar POSIX, incluyendo:

- Sintaxis `$((...))` para expresiones aritméticas.
- Sintaxis `$(...)` para sustitución de comandos, excepto que la abreviatura `$(< filename)` para `$(cat filename)` no está soportada.
- Expansión de tilde (originalmente derivada del shell C).

Las siguientes características del shell de Korn se dejan «no especificadas» en el estándar, lo que significa que su sintaxis es aceptable pero su funcionalidad no está estandarizada:

- Sintaxis `((...))` para condicionales aritméticos. Sin embargo, se incluyen los operadores de prueba aritmética introducidos en el Capítulo 5 (por ejemplo, `-eq, -lt`).
- La sintaxis `[[...]]` para *pruebas* condicionales. Debe usarse el comando de *prueba* externo o `[...]` en su lugar. La versión del shell de Korn de la prueba cumple con POSIX cuando se usa con no más de tres argumentos. (También cumple con cuatro argumentos, si el primer argumento es !).
- La sintaxis para definir funciones que usa este libro. La otra sintaxis mostrada en el Capítulo 4 (`fname()` en lugar de `function fname`) se admite, con lo que describimos como «semántica POSIX»; ver más abajo.
- La estructura de control `select`.
- Solo se permiten números de señal si los números para ciertas señales clave (INT, TERM y algunas otras) son iguales que en las versiones históricas más importantes de Unix. En general, los scripts de shell deben usar nombres simbólicos para las señales.

El estándar POSIX admite funciones, pero la semántica es más débil que la de las funciones de estilo de `function` del shell de Korn: las funciones no tienen trampas u opciones locales, y no es posible definir variables locales. (Por esta razón, *ksh93* tiene dos sintaxis diferentes para definir funciones, con semánticas diferentes).

Se admiten bloques de código (`{...;}`). Para obtener máxima portabilidad, cuando desees llaves literales, debes entrecomillarlas (por razones demasiado complicadas para entrar aquí).

El estándar POSIX introdujo las siguientes características, que difieren del comportamiento tradicional del shell de Bourne. *ksh93* las admite todas:

- Se cambió el orden de búsqueda de comandos para permitir que ciertos comandos

integrados sean anulados por funciones, ya que los alias no están incluidos en el estándar. Los comandos integrados se dividen en dos conjuntos según su posición en el orden de búsqueda de comandos: algunos se procesan antes que las funciones, otros después. Específicamente, los comandos integrados *break*, : (no hacer nada), *continue*, . (dot), *eval, exec, exit, export, readonly, return, set, shift, trap,* y *unset* tienen prioridad sobre las funciones.

- Un nuevo comando integrado, *command*, te permite usar comandos integrados que no están en la lista anterior incluso si hay funciones con el mismo nombre. [4]
- Una nueva palabra clave, !, toma la negación lógica del estado de salida de un comando: si *command* devuelve un estado de salida 0, `! command` devuelve 1; si *comand* devuelve un valor no nulo, `! command` devuelve 0. ! se puede usar con `&&`, ||, y paréntesis (para subcadenas anidadas) para crear combinaciones lógicas de estados de salida en condicionales.
- El comando `unset -v` se utiliza en lugar de `unset` (sin una opción) para eliminar la definición de una variable. Esto proporciona una mejor coincidencia sintáctica con `unset -f`, para deshacer la definición de funciones.

Finalmente, debido a que el estándar POSIX está destinado a promover la portabilidad de scripts de shell, evita mencionar explícitamente características que solo se aplican al uso interactivo del shell, incluidos alias, modos de edición, teclas de control, y así sucesivamente. La opción de Utilidades de Portabilidad del Usuario cubre estos. También evita mencionar ciertos problemas clave de implementación: en particular, no hay requisito de que se use la multitarea para trabajos en segundo plano, subcadenas, etc. Esto se hizo para permitir la portabilidad a sistemas no multitarea como MS-DOS, de modo que, por ejemplo, el Toolkit MKS (ver más adelante en este apéndice) pueda ser compatible con POSIX.

A.4. dtksh

El Desk Top Korn Shell (*dtksh*) es una parte estándar del *Common Desktop Environment (CDE)*, disponible en sistemas Unix comerciales como Solaris, HP-UX y AIX. Está basado en una versión algo más antigua de *ksh93*. Evolucionó a partir del programa anterior *wksh*, el Windowing shell de Korn, lanzado por Unix System Laboratories a fines de 1992. Es

[4] Pero ten en cuenta que no es un comando integrado especial. Aquí se nota el diseño por comité.

un shell de Korn completo, compatible con la versión que describe este libro,[5] y tiene extensiones para la programación de interfaces gráficas de usuario (GUI) en el entorno del sistema X Window. Se encuentra típicamente en `/usr/dt/bin/dtksh`.

dtksh admite el Toolkit gráfico OSF/Motif al poner sus rutinas a disposición como comandos integrados. Esto permite a los programadores combinar la fortaleza del shell de Korn como un entorno de programación de sistemas Unix con el poder y la abstracción del Toolkit. El resultado es un entorno unificado para el desarrollo rápido y sencillo de software basado en gráficos.

Existen varias herramientas de desarrollo de GUI que permiten construir interfaces de usuario con un editor basado en gráficos en lugar de con código de lenguaje de programación. Pero tales herramientas suelen ser enormes, costosas y complejas. *dtksh*, por otro lado, es económico e insuperable en cuanto a su integración suave con Unix; ¡es la única herramienta de este tipo que puedes usar como tu shell de inicio de sesión! (Bueno, casi; consulta la siguiente sección). Es una opción definitiva para los programadores de sistemas que utilizan estaciones de trabajo basadas en X y necesitan una herramienta de prototipado rápido.

Para darte una idea del código *dtksh*, aquí tienes un script que implementa el programa canónico «Hola, mundo» mostrando una pequeña ventana con un botón «Aceptar». Es del artículo *Graphical Desktop Korn Shell*, en el número de julio de 1998 de Linux Journal, escrito por George Kraft IV. (Ver http://www.linuxjournal.com/article.php?sid=2643.) Este código no debería tener sorpresas para los programadores de X y Motif:

```
#!/usr/dt/bin/dtksh

XtInitialize TOPLEVEL dtHello DtHello "$@"
XmCreateMessageDialog HELLO $TOPLEVEL hello \
    dialogTitle:"DtHello" \
    messageString:"$(print "Hola\nMundo")"
XmMessageBoxGetChild HELP $HELLO \
    DIALOG_HELP_BUTTON
XtUnmanageChild $HELP
XmMessageBoxGetChild CANCEL $HELLO \
    DIALOG_CANCEL_BUTTON
XtUnmanageChild $CANCEL
XtAddCallback $HELLO okCallback exit
XtManageChild $HELLO
XtMainLoop
```

[5] Las características enumeradas a lo largo del libro como introducidas en versiones «recientes» no estarán en *dtksh*.

http://www.cactus.org/~gk4/kraft/george/papers/dtksh/ es la presentación web del Sr. Kraft sobre *dtksh*.

El siguiente libro está dedicado a *dtksh*: *Desktop KornShell Graphical Programming* de J. Stephen Pendergast, Jr., publicado por Addison-Wesley, 1995 (ISBN: 0-201-63375-2). Los ejemplos del libro están disponibles en línea en ftp://ftp.aw.com/aw.prof.comp.series/pendergrast.examples.tar.Z. También está disponible ftp://ftp.aw.com/aw.prof.comp.series/pend.dtksh1, un archivo de texto que proporciona una descripción general del libro. Lamentablemente, al momento de escribir esto, este libro está fuera de impresión.

A.5. tksh

En 1996, mientras era estudiante de posgrado en informática en la Universidad de Princeton, el Dr. Jeffrey L. Korn [6] escribió *tksh*. Esta es una integración de *ksh93* con Tcl/Tk. La siguiente cita (de la página web de investigación del Dr. Korn) lo resume bien:

Tksh es un lenguaje gráfico (similar a Visual Basic o Tcl/Tk) que utiliza KornShell (*ksh93*) para la creación de scripts y Tk para la interfaz gráfica de usuario. *Tksh* está implementado como una extensión de *ksh93* y permite que las bibliotecas de Tcl, como Tk, se ejecuten sobre *ksh93* sin cambios. *ksh93* es adecuado para la creación de scripts gráficos porque es compatible con *sh*, lo que facilita tanto el aprendizaje como la extensión de scripts existentes para proporcionar una interfaz de usuario. *Tksh* también permite que los scripts de Tcl se ejecuten sin modificaciones, lo que hace posible combinar componentes escritos en Tcl o *ksh93*.

La página de inicio de *tksh* todavía está en Princeton: http://www.cs.princeton.edu/~{}jlk/tksh/. Tiene enlaces a documentos y documentación que se pueden descargar e imprimir. Sin embargo, el enlace a los ejecutables de *tksh* está desactualizado. El código fuente de *tksh* está disponible en AT&T Research como parte del paquete ast-open, que también contiene *ksh93* y reimplementaciones de muchas otras herramientas de Unix. Consulta el Apéndice C para obtener más información.

El siguiente script de ejemplo, del artículo de USENIX sobre *tksh*, se llama *watchdir*:

```
# Demo de Tksh
# Jeff Korn
#
# Este script realiza un seguimiento de los directorios visitados y muestra los archivos
```

[6] Sí, el hijo de David Korn. Ahora trabaja en el mismo centro de investigación que su padre en AT&T Laboratories, aunque en un área diferente.

```
# en el directorio actual. Puedes hacer doble clic en archivos y
# directorios. El script debe usarse de manera interactiva, así que para ejecutar:
#   $ tksh
#   $ . scripts/watchdir

function winsetup {
  pack $(frame .f)
  frame .f.dirname -relief raised -bd 1
  pack .f.dirname -side top -fill x
  pack $(frame .f.ls) $(frame .f.dirs) -side left
  label .f.dirname.label -text "Directorio actual: "
  label .f.dirname.pwd -textvariable PWD
  pack .f.dirname.label .f.dirname.pwd -side left

  scrollbar .f.ls.scroll -command ".f.ls.list yview"
  listbox .f.ls.list -yscroll ".f.ls.scroll set" -width 20 -setgrid 1
  pack $(label .f.ls.label -text "Contenido del directorio") -side top
  pack .f.ls.list .f.ls.scroll -side left -fill y -expand 1

  scrollbar .f.dirs.scroll -command ".f.dirs.list yview"
  listbox .f.dirs.list -yscroll ".f.dirs.scroll set" -width 20 -setgrid 1
  pack $(label .f.dirs.label -text "Directorios visitados") -side top
  pack .f.dirs.list .f.dirs.scroll -side left -fill y -expand 1
  bind .f.dirs.list "" 'cd $(selection get)'
  bind .f.ls.list "" 'tkfileselect $(selection get)'
}

function tkfileselect {
  [[ -d "$1" ]] && tkcd "$1"
  [[ -f "$1" ]] && ${EDITOR-${VISUAL-emacs}} "$1"
}

function tkcd {
  cd $1 > /dev/null || return
  .f.ls.list delete 0 end
  set -o markdirs
  .f.ls.list insert end .. *
  [[ ${VisitedDir["$PWD"]} == "" ]] && .f.dirs.list insert end "$PWD"
  VisitedDir["$PWD"]=1
}

typeset -A VisitedDir
winsetup > /dev/null
alias cd=tkcd
tkcd .
```

Lo interesante de *tksh*, además de la interesante combinación de tecnologías complementarias, es que lleva la programación Tk al nivel del shell. La programación gráfica con Tk es mucho más elevada que con el kit de herramientas Motif; por lo tanto, la curva de aprendizaje es más fácil de superar y los scripts son más fáciles de leer y escribir.

A.6. pdksh

Muchos de los sistemas tipo Unix de código abierto, como GNU/Linux, vienen con el Public Domain Korn Shell, *pdksh*. *pdksh* está disponible como código fuente; comienza en su página de inicio: http://www.cs.mun.ca/~{}michael/pdksh/. Incluye instrucciones para la construcción e instalación en varias plataformas Unix.

pdksh fue originalmente escrito por Eric Gisin, quien lo basó en el clon de dominio público del shell de Bourne de la Versión 7 creado por Charles Forsyth. Es en su mayoría compatible con el shell de Korn de 1988 y POSIX, con algunas extensiones propias.

Su modo de edición en *emacs* es en realidad más potente que el del shell de Korn de 1988. Al igual que el editor Emacs completo, puedes personalizar las teclas que invocan comandos de edición (conocidos como *key bindings* en la terminología de Emacs). Los comandos de edición tienen nombres completos que puedes asociar con teclas mediante el uso del comando *bind*.

Por ejemplo, si quieres configurar CTRL-U para hacer lo mismo que CTRL-P (es decir, retroceder al comando anterior en el archivo de historial), podrías poner este comando en tu *.profile*:

```
bind '^U'=up-history
```

Incluso puedes configurar secuencias de escape de dos caracteres, que (por ejemplo) te permiten usar las teclas de flecha ANSI además de caracteres de control, y puedes definir *macros*, es decir, abreviaturas para secuencias de comandos de edición.

Las características adicionales del Public Domain Korn Shell incluyen comodines de *alternancia* (tomados del C shell) y notación de tilde definible por el usuario, en la que puedes configurar ~ como una abreviatura para cualquier cosa, no solo nombres de usuario. También hay algunas diferencias sutiles en la evaluación de expresiones enteras y en la creación de alias.

pdksh carece de las siguientes características de la versión oficial:

- La variable integrada `LINES`.
- La señal falsa `DEBUG`.
- Las señales falsas `ERR` y `EXIT` dentro de funciones.
- Las funciones heredan la configuración de traps del script principal.

- Las clases de caracteres de expansión de archivos POSIX (`[[:alpha:]]`, etc.) no están disponibles.
- El comando *read* y el bucle *select* no utilizan los modos de edición de línea de comandos.
- El último comando de una canalización no se ejecuta en el shell principal. Por lo tanto, `echo hi | read x; print $x` no funciona igual que en *ksh*. (La mayoría de los shells de estilo Bourne funcionan de esta manera).
- La forma de opción `set -o` de *ksh* es `set -X option` en *pdksh*.

Aunque carece de la mayoría de las extensiones de *ksh93*, *pdksh* es una alternativa valiosa a los shells C y Bourne.

A.7. bash

bash es probablemente el shell «tercerizado» más popular disponible para Unix y otros sistemas. En particular, es el shell predeterminado en sistemas GNU/Linux (también se incluye en el CD «freeware» con Solaris 8). Puedes obtenerlo desde Internet, a través de FTP anónimo en ftp.gnu.org en el directorio `/pub/gnu/bash`. También puedes pedirlo a su fabricante en la dirección que se indica aquí.

The Free Software Foundation

59 Temple Place - Suite 330

Boston, MA 02111-1307

(617) 542-2652

(617) 542-5942 (fax)

gnu@gnu.org

http://www.gnu.org

Bash fue escrito por Brian Fox y Chet Ramey. Chet Ramey es el actual mantenedor. Su nombre está en línea con la predilección de la FSF por los juegos de palabras: significa Bourne-Again Shell. Aunque *bash* está fácilmente disponible y no tienes que pagar por ello (además del costo de los medios, llamadas telefónicas, etc.), no es realmente un software de dominio público. Mientras que el software de dominio público no tiene restricciones de licencia, el software de la FSF sí las tiene. Pero esas restricciones son diametralmente

opuestas a las de una licencia comercial:[7] ¡en lugar de acordar no distribuir el software más, acuerdas no impedir que se distribuya más! En otras palabras, disfrutas del uso ilimitado del software siempre y cuando aceptes no obstaculizar que otros hagan lo mismo. Richard Stallman, el fundador de la FSF, inventó este concepto intrigante y admirable.

Hoy en día, los ideales de los movimientos de Software Libre y Código Abierto, el proyecto GNU y la calidad del software GNU son bien conocidos. El sistema GNU más popular es GNU/Linux, que utiliza el núcleo Linux y las utilidades GNU para crear un entorno de computación completo, totalmente funcional y compatible con Unix y POSIX.

bash es totalmente compatible con el estándar POSIX de 1992. Tiene varias de las características más importantes del shell de Korn y las características del C shell que el shell de Korn ha adoptado, incluyendo alias, funciones, notación de tilde, modos de edición emacs y vi, expresiones aritméticas, control de trabajos, etc.

La superposición de características entre *bash* y el shell de Korn ha aumentado en los últimos años. Incluye muchas características del *ksh93*. Pero no es un clon exacto de *ksh*. El FAQ de *bash*, publicado mensualmente por Chet Ramey, enumera las siguientes diferencias entre *bash* y *ksh93*. Los elementos incluidos entre corchetes (`[...]`) están listados en este libro, pero no en el FAQ.

Lo siguiente, en *ksh93*, no está disponible en *bash* 2.05:

- Arrays asociativos
- Aritmética y variables de punto flotante
- Funciones de la biblioteca matemática `${!nombre[sub]}` nombre de subíndice para arreglo asociativo
- `"."` está permitido en nombres de variables para crear un espacio de nombres jerárquico
- Sintaxis de asignación compuesta más extensa
- Funciones de disciplina
- Instrucciones integradas *sleep* y *getconf* (*bash* tiene versiones cargables)
- `typeset -n` y variables *nameref*
- La trampa KEYBD

[7] En consecuencia, el documento que detalla estas restricciones se llama *copyleft*.

- Las variables: `.sh.edchar, .sh.edmode, .sh.edcol, .sh.edtext, .sh.version, .sh.name, .sh.subscript, .sh.value, HISTEDIT` [La variable `.sh.match` también]
- Referencias posteriores en la coincidencia de patrones (\N)
- El operador & en listas de patrones para coincidencia
- `print -f` (bash usa *printf*)
- *fc* ha sido renombrado a *hist*
- El comando de punto (.) puede ejecutar funciones de shell
- Estados de salida entre 0 y 255
- La opción de asignación de variables +=
- `TMOUT` es el tiempo de espera predeterminado para *read* y *select*
- Redirecciones `<&n-` y `>&n-` (combinación de duplicar y cerrar) [Aquí-strings con `<<<`]
- Mezcla de `FPATH` y `PATH`
- `getopts -a`
- La opción de invocación `-R`
- La trampa `DEBUG` ahora se ejecuta antes de cada comando simple, en lugar de después
- Los modificadores `%H, %P, %T` de *printf*, y una base de salida para `%d` [También `%Z`.]

Lo siguiente, en *ksh93*, está presente en *bash* 2.05:

- El comando `for ((...;...;...)) ; do list; done` para comandos aritméticos
- Los operadores aritméticos `?:, ++, --` y coma (`,`)
- Las expansiones de variables de shell: `${!param}, ${param:offset[:len]}, ${param/pat[/str]` `${!param*}`
- Asignación compuesta de arreglos
- La palabra reservada !
- Instrucciones integradas cargables, pero *ksh* usa *builtin* mientras que *bash* usa *enable* Las instrucciones integradas *command*, *builtin* y *disown*
- La citación nueva *$'...'* y *$"..."*

- `FIGNORE` (pero *bash* usa `GLOBIGNORE`), `HISTCMD`
- `set -o notify`, `set -C`
- Cambios en el comando integrado *kill*
- `read -A` (*bash* usa `read -a`)
- `read -t/read -d`
- `trap -p`
- `exec -a/exec -c`
- El comando de punto (.) restaura los parámetros posicionales cuando se completa
- El comando test se ajusta a POSIX.
- `umask -S`
- `unalias -a`
- Sustitución de comandos y aritmética realizada en `PS1, PS4` y `ENV`
- Completado de nombres de comandos
- `ENV` procesado solo para shells interactivos

bash tiene muchas características propias que lo convierten en un entorno muy potente y flexible. Aquí tienes algunas de las características destacadas:

- Puedes poner escapes de barra invertida en la cadena de texto del indicador primario (`PS1`), en la que *bash* sustituye cosas como la fecha, hora, directorio de trabajo actual, nombre de la máquina, nombre de usuario, shell, etc.
- Los comandos integrados, *command* y *enable*, te brindan más control sobre los pasos que sigue *bash* para buscar comandos, es decir, el equivalente de *bash* a la lista de pasos de búsqueda de comandos en el Capítulo 7.
- El modo de edición *emacs* es personalizable, incluso más que su equivalente en *pdksh*. Puedes usar el comando *bind* para configurar tus propias preferencias de teclas, y hay varios comandos más disponibles, incluida la capacidad de deshacer tu último comando.
- También puedes volver a asignar teclas en el modo de edición *vi*.
- *pushd* y *popd* están integrados, al igual que en el C shell.

- Los arreglos indexados pueden tener un tamaño ilimitado.
- Muchas opciones y variables nuevas te permiten personalizar tu entorno con una flexibilidad sin precedentes. Esto incluye `set -o posix` para una conformidad estricta con POSIX.

Nos sentimos obligados a decir que muchos usuarios prefieren *bash* al shell de Korn. Con la creciente popularidad de GNU/Linux y varios sistemas derivados de BSD, no está claro qué shell tiene una base de usuarios más grande. En cualquier caso, *bash* es definitivamente una excelente elección. Recomendamos el libro *Learning the bash Shell* de Cameron Newham y Bill Rosenblatt, publicado por O'Reilly & Associates. (Está basado en la primera edición de este libro).

A.8. zsh

zsh es un potente shell interactivo y lenguaje de secuencias de comandos con muchas características que se encuentran en *ksh, bash* y *tcsh*, así como varias características únicas. *zsh* tiene la mayoría de las características de *ksh88* pero pocas de *ksh93*. Está disponible de forma gratuita y debería compilar y ejecutarse en prácticamente cualquier versión moderna de Unix. También hay versiones para otros sistemas operativos. La página principal de *zsh* es http://www.zsh.org. La versión actual es 4.0.2.

En esta sección cubrimos:

- Globbing extendido
- Completado
- Edición de línea de comandos
- Indicadores y temas de indicadores
- Diferencias entre *zsh* y *ksh*

A.8.1. Globbing extendido

Una característica muy útil es el operador emphglob recursivo `**`[8] . Por ejemplo, es sencillo construir una búsqueda recursiva con *grep*:

```
grep foo **/*.c
```

[8]El *globbing* es un argot técnico para la expansión de comodines.

O para encontrar recursivamente todos los archivos o directorios llamados *core*, prueba:

```
print **/core
```

Otra característica útil son los *calificadores de glob*. Hay muchos de ellos, por ejemplo, para imprimir solo archivos regulares en el directorio actual:

```
print *(.)
```

o solo los directorios:

```
print *(/)
```

Combinar estos con el operador glob recursivo puede ser útil. Podemos mejorar el ejemplo anterior de encontrar archivos *core* limitando la búsqueda solo a archivos regulares:

```
print **/core(.)
```

Otro calificador es U para objetos del sistema de archivos de los que eres propietario. Lo siguiente imprime todos los archivos que posees en `/tmp` y sus subdirectorios:

```
print /tmp/**/*(U)
```

Los calificadores de glob también se pueden combinar. Por ejemplo, usando la palabra clave del archivo de socket = en combinación con U, es fácil encontrar archivos de socket en `/tmp` y sus subdirectorios que posees:

```
print /tmp/**/*(U=)
```

También están disponibles calificadores de tamaño de archivo. Por ejemplo, para encontrar todos los archivos en tu directorio de inicio que tienen más de 10 megabytes de tamaño:

```
print ~/**/*(Lm+10)
```

Y también hay calificadores de permisos de archivo. Por ejemplo, el calificador W selecciona objetos del sistema de archivos que son escribibles por el mundo. Puedes usarlo para encontrar todos los directorios en tu directorio de inicio y sus subdirectorios que te pertenecen y que son escribibles por el mundo:

```
print ~/**/*(UW/)
```

Consulta *zshexpn(1)* para obtener más información.

A.8.2. Completado

El sistema de completado de *zsh* es extremadamente sofisticado. La idea principal es que cada vez que estás a punto de escribir algo en la línea de comandos, si presionas TAB, *zsh*

intentará completarlo. *zsh* viene con muchos valores predeterminados para la completación y también es completamente personalizable.

Para obtener un conjunto completo de funciones de completado predeterminadas, ejecuta los siguientes comandos (normalmente en tu archivo de inicio `/.zshrc`):

```
autoload -U compinit
compinit
```

Ahora veamos algunos ejemplos. Representamos la tecla TAB en los ejemplos como `[TAB]`.

Primero, *zsh* es inteligente al realizar completados. Por ejemplo, `cd[TAB]` solo expande directorios, eliminando así el ruido de la completación.

¿Alguna vez te has sentido frustrado porque no puedes recordar exactamente el nombre del comando del que deseas obtener más información, y *man -k*[9] no está configurado en tu sistema? Bueno, *zsh* completará las páginas del manual disponibles para ti:

```
g@sifl:pts/7% man zsh[TAB]
zsh           zshcompctl    zshcontrib    zshmodules     zshzftpsys
zshall        zshcompsys    zshexpn       zshoptions     zshzle
zshbuiltins   zshcompwid    zshmisc       zshparam
```

O tal vez quieras descubrir un nombre de proceso o PID que deseas matar:

```
g@sifl:pts/2% kill [TAB]
9652    pts/2      00:00:00  zsh
9653    pts/2      00:00:00  ps
```

Para *finger*, expande usuarios:

```
g@sifl:pts/7% finger o[TAB]
odin    omg    oof    operator    orb
```

y hosts:

```
g@sifl:pts/7% finger oof@[TAB]
brak    localhost    sifl    zorak
```

Usando la función *compdef* distribuida, puedes definir tus propias completaciones, utilizando tus propias funciones personalizadas o las funciones de completado que vienen con *zsh*. Por ejemplo, la distribución define el comando *kill* para usar la función de distribución `_pids` para proporcionar identificadores de procesos. También puedes usarlo para definir la completación para otros comandos, como el comando *pstack* de Solaris:

```
compdef _pids pstack
```

[9] Esto realiza una búsqueda de palabras clave en una base de datos en línea extraída de las páginas del manual.

Una vez hecho esto, puedes aplicar la completación al comando *pstack* de la siguiente manera:

```
g@sifl:pts/7% pstack [TAB]
13606 pts/7    00:00:00 zsh
13607 pts/7    00:00:00 ps
```

Otra función de completado de distribución muy útil es *_gnu_generic*. Esto se puede aplicar a cualquier comando que utilice las convenciones de opción de línea de comando `--long-option` de GNU. La distribución de *zsh* especifica muchos comandos de GNU para completar con esta función (como *tar*):

```
g@sifl:pts/7% tar --v[TAB]
--verbose   --verify   --version   --volno-file
```

Y esto es solo la punta del proverbial iceberg. Hay mucho más en el sistema de completado de *zsh*; consulta *zshcompsys(1)* para los detalles (*gory*).

A.8.3. Editor de línea de comandos

El editor de línea de comandos de *zsh* es extremadamente potente. Tiene varias características únicas, incluida la edición de varias líneas y una pila de búfer de entrada. El editor de comandos de varias líneas facilita la composición de pequeños scripts en la línea de comandos, ya que no solo tienes una línea para editar.

La pila de búfer de entrada resulta muy útil. Mientras escribes un comando, puedes escribir ESC q y la línea actual se coloca en la pila de búfer. La línea de entrada se borra y puedes escribir otro comando. Cuando se ejecuta ese comando, la línea anterior se saca de la pila y puedes continuar con ese comando. Consulta *zshzle(1)* para obtener más detalles.

A.8.4. Indicadores y temas de indicadores

Aunque la mayoría de las shells modernas tienen indicadores personalizables, *zsh* lo eleva a una forma de arte. Una de las características únicas es un indicador del lado derecho, `RPROMPT`, que es muy útil para mostrar el directorio actual. Esto a su vez elimina el desorden del indicador del lado izquierdo:

```
g@sifl:pts/2% RPROMPT='%~'
g@sifl:pts/2%              ~/src/xemacs-21.1.14
```

Además, puedes definir colores y fuentes en negrita, y el indicador puede ocupar más de una línea.

Y dado que la noción de temas[10] se ha vuelto popular en interfaces de usuario gráficas como GNOME, se pueden definir temas de indicadores de *zsh*; la distribución incluye varios entre los que puedes elegir. Para habilitar los temas de indicadores, agrega estas líneas a tu `~/.zshrc`:

```
autoload -U promptinit
promptinit
```

Para ver qué temas están disponibles, ejecuta:

```
g@sifl:pts/2% prompt -l
Currently available prompt themes:
adam1 adam2 bart bigfade clint elite2 elite fade fire off oliver
redhat suse zefram
```

Para habilitar un tema, usa la opción `-s`. Por ejemplo:

```
g@sifl:pts/7% prompt -s bart
sifl [prompt -s bart] ~                          01-10-04 11:58PM
g@sifl:pts/7%
```

Puedes ver que *bart* es un indicador de dos líneas con varios componentes como el nombre del host, el comando anterior, el directorio actual y la fecha y hora. Consulta *zshcontrib(1)* para obtener más detalles sobre los temas de indicadores.

A.8.5. Diferencias entre *zsh* y *ksh*

Esta sección se deriva de información en el FAQ de *zsh*.

La mayoría de las características de *ksh88* (y, por lo tanto, también del Shell de Bourne, *sh*) están implementadas en *zsh*; pueden surgir problemas porque la implementación es ligeramente diferente. También ten en cuenta que no todas las *ksh* son iguales. Esto se basa en la versión 11/16/88f de *ksh*; las diferencias con *ksh93* son más sustanciales.

Como resumen del estado:

1. Debido a todas las opciones, no es seguro asumir que una ejecución general de *zsh* por un usuario se comportará como *sh* o *ksh* compatible.

2. Invocar *zsh* como *sh* o *ksh* (o si alguno de ellos es un enlace simbólico a *zsh*) establece opciones apropiadas y mejora la compatibilidad (desde dentro de *zsh*, llamar `ARGV0=sh zsh` también funcionará).

[10] Algunas GUIs populares, como GNOME, soportan temas. En lugar de tener una apariencia inmutable, pueden cambiarse a diferentes estilos o temas. Las distribuciones de estos GUIs a menudo contienen varios para elegir. Algunas de ellas tienden a emular otras GUI, mientras que otras son nuevas y son más que nada divertidas decoraciones de ventanas.

3. Desde la versión 3.0 en adelante, el grado de compatibilidad con *sh* en estas circunstancias es muy alto: *zsh* ahora se puede usar con GNU *configure* o *Configure de perl*, por ejemplo.

4. El grado de compatibilidad con *ksh* también es alto, pero faltan algunas cosas: por ejemplo, las expresiones de coincidencia de patrones más sofisticadas son diferentes para versiones anteriores a 3.1.3; consulta la lista detallada a continuación.

5. También a partir de 3.0, está disponible el comando *emulate*: `emulate ksh` y `emulate sh` establecen varias opciones, así como cambian el efecto de las banderas de opciones de un solo carácter, como si el shell se hubiera invocado con el nombre apropiado. Incluir el comando `emulate sh; setopt localoptions` en una función del shell activa la emulación *sh* solo para esa función. En 4.0 (y en 3.0.6 a 8), esto se puede abreviar como `emulate -L sh`.

La diferencia clásica es la división de palabras: *zsh* mantiene el resultado de `$variable` como una sola palabra, incluso si la variable contiene espacios en blanco. Esto confunde a muchos *usuarios principiantes de zsh*. La respuesta es establecer `SH_WORD_SPLIT` para compatibilidad hacia atrás. La siguiente diferencia más clásica es que los patrones de *globs* no coincidentes hacen que el comando *abort*; establece `NO_NOMATCH` para evitar eso.

zsh tiene un conjunto grande de opciones que aumentan la compatibilidad con *ksh*, aunque tal vez disminuyan las habilidades de *zsh*: consulta las entradas del manual para más detalles. Si se invoca como *ksh*, el shell establece las opciones adecuadas.

Aquí hay algunas diferencias con *ksh* que podrían ser significativas para los programadores de *ksh*, algunas de las cuales pueden interpretarse como errores. Ten en cuenta que esta lista está deliberadamente bastante completa y que la mayoría de los elementos son bastante menores. Aquellos marcados con ↾ realizan de manera similar a *ksh* si el shell se invoca con el nombre *ksh* o si está en efecto `emulate ksh`. Las palabras en mayúsculas con guiones bajos en sus nombres se refieren a opciones del shell.

Sintaxis

- ↾ División de palabras en el shell: consulta arriba.
- ↾ Los arrays son (por defecto) más parecidos a *csh* que a *ksh*: los subíndices comienzan en 1, no en 0; `array[0]` se refiere a `array[1]`; `$array` se refiere a todo el array, no a `$array[0]`; las llaves no son necesarias: `$a[1]` es lo mismo que `${a[1]}`, etc. Establece la opción `KSH_ARRAYS` para compatibilidad.

- Los coprocesos se establecen con `coproc; |&` se comporta como *csh*.[11] El manejo de los descriptores de archivo de coproceso también es diferente.
- Para `cmd1 && cmd2 &`, solo `cmd2`, en lugar de toda la expresión, se ejecuta en segundo plano en *zsh*. El manual da a entender que esto es un error. Usa `{ cmd1 && cmd2 } &` como solución.

Substituciones de línea de comandos, globbing, etc.

- ǂ La falta de coincidencia con un patrón de *globbing* provoca un error (usa `NO_NOMATCH`).
- ǂ Los resultados de las sustituciones de parámetros se tratan como texto plano: `foo="*"; print $foo` imprime todos los archivos en *ksh* pero imprime * en *zsh* (usa `GLOB_SUBST`).
- ǂ Las variables de indicador (por ejemplo, `PS1`) no sufren sustitución de parámetros por defecto (usa `PROMPT_SUBST`).
- ǂ El globbing estándar no permite listas de patrones al estilo *ksh*. La Tabla A.1 muestra patrones equivalentes.
  Las formas ^, ~ y # (pero no |) requieren establecer `EXTENDED_GLOB`. A partir de la versión 3.1.3, las formas de *ksh* son completamente compatibles cuando está en efecto la opción `KSH_GLOB`; para versiones anteriores debes usar los equivalentes dados en la Tabla A.1.
- Las asignaciones no entrecomilladas hacen la expansión de archivos después de los dos puntos (destinado para variables de estilo `PATH`).
- *integer* no permite `-i`.
- *typeset* e *integer* tienen un comportamiento especial para las asignaciones en *ksh*, pero no en *zsh*. Por ejemplo, esto no funciona en *zsh*:

  ```
  integer k=$(wc -l ~/.zshrc)
  ```

  porque el valor de retorno de *wc* incluye espacios en blanco iniciales, lo que provoca la división de palabras. *ksh* maneja la asignación de manera especial, como una sola palabra.

[11] En `csh, |&` envía tanto la salida estándar como la salida de error por el mismo conducto; es equivalente a ... `2>&1 | ...`.

[12] Ten en cuenta que ~ es el único operador de *globbing* que tiene una precedencia inferior a /. Por ejemplo, `**/foo~*bar*` coincide con cualquier archivo en un subdirectorio llamado *foo*, excepto donde *bar* ocurrió en alguna parte de la ruta (por ejemplo, `users/barstaff/foo` será excluido por el operador ~). Como el operador `**` no puede agruparse (dentro de paréntesis se trata como *), esta es la manera de

Tabla A.1: Equivalentes de patrones *ksh/zsh*

ksh	**zsh**	**Significado**
!(foo)	^foo	Cualquier cosa menos *foo*
	foo1~foo2	Cualquier cosa que coincida con foo1 pero no con foo2.[12]
@(foo1|foo2|...)	(foo1|foo2|...)	*foo1* o *foo2* o ...
?(foo)	(foo|)	Cero o una ocurrencia de *foo*
*(foo)	(foo)#	Cero o más ocurrencias de *foo*
+(foo)	(foo)##	Uno o más ocurrencias de *foo*

Ejecución de comandos

- ↾ No hay variable `ENV` (usa `/etc/zshrc`, `~/.zshrc`; también nota `$ZDOTDIR`).
- `$PATH` no se busca para comandos especificados en la invocación sin `-c`.

Alias y funciones

- El orden en que se definen los alias y las funciones es significativo: las definiciones de funciones con () expanden alias.
- Los alias y las funciones no se pueden exportar.
- No hay alias rastreados: el hashing de comandos reemplaza esto.
- El uso de alias para la vinculación de teclas se reemplaza por *bindkey*.
- ↾ Las opciones no son locales para las funciones (usa `LOCAL_OPTIONS`; nota que esto siempre puede no estar configurado localmente para propagar la configuración de opciones desde una función hasta el nivel de llamada).
- Las funciones definidas con `function funcname { body ;}` se comportan de la misma manera que las definidas con `funcname () { body ;}`. En *ksh93*, solo las primeras se comportan como funciones verdaderas, y las segundas se comportan como si el cuerpo se leyera desde un archivo con el comando punto.

Trampas y señales

- ↾ Las trampas no son locales para las funciones. La opción `LOCAL_TRAPS` está disponible desde 3.1.6.
- `TRAPERR` se convierte en `TRAPZERR` (esto fue forzado por `UNICOS` que tiene `SIGERR`).

Edición

excluir algunos subdirectorios de hacer coincidir un **.

- Las opciones *emacs, gmacs* y *viraw* no son compatibles. Usa *bindkey* para cambiar el comportamiento de edición: `set -o emacs` se convierte en `bindkey -e` y `set -o vi` se convierte en `bindkey -v`; para *gmacs*, ve a *emacs-moda* y usa `bindkey \^t gosmacs-transpose-characters`.
- La opción de *palabra clave* no existe y `set -k` es en su lugar *interactivecomments*.
- ↾ La gestión de historiales en múltiples shells es diferente: la lista de historiales no se guarda y restaura después de cada comando. La opción `SHARE_HISTORY` apareció en 3.1.6 y se establece en el modo de compatibilidad con *ksh* para remediar esto.
- \ no escapa caracteres de edición (usa CTRL-V).
- No se establecen todos los enlaces de *ksh* (por ejemplo, ESC #; prueba ESC q).
- ↾ # en una shell interactiva no se trata como un comentario por defecto.

Comandos incorporados

- Algunos comandos incorporados (*r, autoload, history, integer, ...*) son alias en *ksh*.
- No hay un comando incorporado *newgrp*: usa `alias newgrp=.^exec newgrp"`.
- *jobs* no tiene una bandera `-n`.
- *read* no tiene una bandera `-s`.

Otras idiosincrasias

- `select` siempre vuelve a mostrar la lista de selecciones en cada bucle.

A.9. Sustitutos en Plataformas PC

La proliferación del shell de Korn no se detuvo en los límites de Unix. Muchos programadores que obtuvieron su experiencia inicial en sistemas Unix y posteriormente cruzaron al mundo de PC deseaban un entorno agradable tipo Unix (especialmente cuando se enfrentaban a los horrores de la línea de comandos MS-DOS), así que no sorprende que hayan aparecido varias interfaces tipo shell Unix para sistemas operativos de pequeñas computadoras, entre ellas, emulaciones del shell de Korn.

En los últimos años, no solo han aparecido clones de shell, sino entornos Unix «completo». Dos de ellos utilizan shells que ya hemos discutido. Dos proporcionan sus propias

re-implementaciones del shell. Proporcionar listas de diferencias mayores y menores es contraproducente. En cambio, esta sección describe cada entorno a su vez (en orden alfabético), junto con información de contacto y descarga en Internet.

A.9.1. Cygwin

Cygnus Consulting (ahora Red Hat) creó el entorno *Cygwin*. Crearon primero *cgywin.dll*, una biblioteca compartida que proporciona emulación de llamadas al sistema Unix, y luego trasladaron un gran número de utilidades GNU a varias versiones de Microsoft Windows. La emulación incluye redes TCP/IP con la API de sockets de Berkeley. La mayor funcionalidad se encuentra en Windows/NT, Windows 2000 y Windows XP, aunque el entorno puede funcionar en Windows 95/98/ME también.

El entorno *Cygwin* utiliza *bash* como su shell, GCC como su compilador C y el resto de las utilidades GNU para su conjunto de herramientas Unix. Un comando de montaje sofisticado proporciona un mapeo de la notación de ruta de Windows `C:\path` a nombres de archivos Unix.

El punto de partida para el proyecto *Cygwin* es http://www.cygwin.com. Lo primero que debes descargar es un programa instalador. Al ejecutarlo, eliges qué paquetes adicionales deseas instalar. La instalación es completamente basada en Internet; no hay CD oficiales de *Cygwin*, al menos no de los mantenedores del proyecto.

A.9.2. DJGPP

El conjunto DJGPP proporciona herramientas GNU de 32 bits para el entorno MS-DOS. Para citar la página web:

- DJGPP es un sistema de desarrollo C/C++ completo de 32 bits para PC Intel 80386 (y superiores) que ejecutan MS-DOS. Incluye puertos de muchas utilidades de desarrollo GNU. Las herramientas de desarrollo requieren una computadora 80386 o más nueva para ejecutarse, al igual que los programas que producen. En la mayoría de los casos, los programas que produce se pueden vender comercialmente sin licencia ni regalías.
- El nombre proviene de las iniciales de D.J. Delorie, quien portó el compilador GNU C++, g++, a MS-DOS, y las iniciales de texto de g++, GPP. Se convirtió esencialmente en un entorno Unix completo en la parte superior de MS-DOS, con todas las

herramientas GNU y *bash* como su shell. A diferencia de *Cygwin* o UWIN (ver más adelante en este capítulo), no necesitas una versión de Windows, solo un procesador de 32 bits completo y MS-DOS. (Aunque, por supuesto, puedes usar DJGPP desde una ventana MS-DOS de Windows). El sitio web es http://www.delorie.com/djgpp/.

A.9.3. MKS Toolkit

Quizás el entorno Unix más establecido para el mundo de las PC es el MKS Toolkit de Mortice Kern Systems:

MKS Canada - Sede Corporativa

410 Albert Street

Waterloo, ON N2L 3V3

Canadá

(519) 884-2251

(519) 884-8861 (fax)

(800) 265-2797 (ventas)

http://www.mks.com

MKS Toolkit viene en varias versiones dependiendo del entorno de desarrollo y la cantidad de desarrolladores que lo utilizarán. Incluye un shell que cumple con POSIX, junto con casi todas las características del shell de Korn de 1988, así como más de 300 utilidades como *awk, perl, vi, make,* y demás. Su biblioteca admite más de 1500 API de Unix, haciéndola extremadamente completa y facilitando la portabilidad al entorno Windows. Más información está disponible en http://www.mkssoftware.com/products/tk/ds_tkpdev.asp.

A.9.4. Thompson Automation Software Toolkit

Thompson Automation Software proporciona el Thompson Toolkit, que incluye un shell y más de 100 utilidades. El kit está disponible para MS-DOS 2.1 y superior, OS/2 1.2 o WARP y para Microsoft Windows 95 y superior. La información de contacto es:

Thompson Automation Software

5616 SW Jefferson

Portland, OR 97221

1-800-944-0139 (EE. UU. y Canadá)

1-503-224-1639 (internacional/local)

1-503-224-3230 (fax)

sales@tasoft.com

http://www.tasoft.com/toolkit.html/

El software de Thompson es conocido por su implementación de *awk*, que es rápida y confiable, con muchas extensiones poderosas al lenguaje awk. El shell del toolkit es compatible con POSIX y la versión 1988 del shell de Korn.

A.9.5. AT&T UWIN

El paquete UWIN es un proyecto de David Korn y sus colegas para poner un entorno Unix disponible bajo Microsoft Windows. Es similar en estructura a *Cygwin*, discutido anteriormente. Una biblioteca compartida, *posix.dll*, proporciona emulación de las APIs de llamadas al sistema Unix. La emulación de llamadas al sistema es bastante completa. Un giro interesante es que el registro de Windows se puede acceder como un sistema de archivos bajo `/reg`. Sobre la emulación de la API de Unix, se han compilado y ejecutado *ksh93* y más de 200 utilidades Unix (o más bien, re-implementaciones). El entorno UWIN depende del compilador nativo de Microsoft Visual C/C++, aunque las herramientas de desarrollo GNU están disponibles para descargar y usar con UWIN.

http://www.research.att.com/sw/tools/uwin/ es la página web del proyecto. Describe lo que está disponible, con enlaces para descargar binarios, así como información sobre la licencia comercial del paquete UWIN. También se incluyen enlaces a varios documentos sobre UWIN, software adicional útil y enlaces a otros paquetes similares.

La ventaja más notable del paquete UWIN es que su shell es el auténtico *ksh93*. Por lo tanto, la compatibilidad con la versión Unix de *ksh93* no es un problema.

CAPÍTULO B

INFORMACIÓN DE REFERENCIA

Este apéndice contiene listas de referencia para opciones de invocación, comandos incorporados y palabras clave, alias predefinidos, variables de shell incorporadas, operadores de *prueba*, opciones de shell, opciones de *typeset*, aritmética, comandos en modo *emacs* y comandos de control en modo *vi*. Además, describe cómo utilizar todas las facilidades del comando incorporado *getopts*.

B.1. Opciones de invocación

Esta es una lista de las opciones que puede usar cuando invoque al shell de Korn. Además de éstas, se puede utilizar cualquier opción establecida en la línea de órdenes; consulte la tabla de opciones más adelante en este apéndice. Los intérpretes de comandos de inicio de sesión suelen invocarse con las opciones `-i` (interactivo), `-s` (leer de la entrada estándar) y `-m` (habilitar el control de trabajos).

Opción	**Significado**
-c *string*	Ejecuta *string* y después sale
-D	Imprime todas las cadenas `$"..."` del script. Esto se utiliza para crear una base de datos de traducciones específicas de cada idioma de las cadenas de un script.
-i	Shell interactivo. Ignora las señales TERM, INTR y QUIT.
-r	Shell restringido. Ver Capítulo 10
-R *filename*	Crea una base de datos de referencias cruzadas para definiciones de variables y comandos en *filename*. No se puede compilar.
-s	Leer comandos de la entrada estándar. Si se proporciona un argumento, esta bandera tiene prioridad (es decir, el argumento no se tratará como un nombre de script y se leerá la entrada estándar).

B.2. Comandos integrados y palabras clave

He aquí un resumen de todos los comandos y palabras clave incorporados.

Nombre	**Comando / Palabra Clave**	**Capítulo**	**Resumen**
!	Palabra clave	5	Invierte el resultado verdadero/falso de la siguiente tubería.
:	Comando	7	No hacer nada (sólo hacer expansiones de argumentos).
.	Comando	4	Leer archivo y ejecutar su contenido en el shell actual.
alias	Comando	3	Establecer la abreviatura de comando o línea de comandos.
bg	Comando	8	Poner trabajo en segundo plano.
builtin	Comando		Añade o elimina comandos incorporados; imprime información sobre ellos.
break	Comando	5	Salir del bucle `for`, `select`, `while` o `until` circundante.
case	Comando	5	Construcción condicional multidireccional.
cd	Comando	1	Cambiar el directorio de trabajo.
command	Comando	5	Localizar comandos integrados y externos; encontrar un comando integrado en lugar de una función con el mismo nombre.
continue	Comando	4	Salta a la siguiente iteración del bucle `for`, `select`, `while` o `until`.
disown	Comando	8	Desasociar un trabajo en segundo plano del shell actual. El efecto es que el trabajo no recibe la señal HUP cuando el shell sale.
echo	Comando	4	Expandir e imprimir argumentos (obsoleto).
exec	Comando	9	Sustituye el shell por el programa dado.

Nombre	**Comando / Palabra Clave**	**Capítulo**	**Resumen**
exit	Comando	5	Salida del shell.
export	Comando	3	Crear variables de entorno.
eval	Comando	7	Procesa los argumentos como una línea de comandos.
false	Comando	8	No hacer nada y salir sin éxito. Útil para hacer bucles infinitos.
fg	Comando	8	Poner el trabajo de fondo en primer plano.
for	Palabra clave	5	Construcción en bucle.
function	Palabra clave	4	Definir función
getconf	Comando		Obtener información específica del sistema. Los parámetros están definidos por POSIX.
getopts	Comando	6	Opciones de la línea de comandos del proceso.
hist	Palabra clave	2	Trabajar con el historial de comandos.
if	Comando	1	Construcción condicional.
jobs	Comando	1	Enumera los trabajos de fondo.
kill	Comando	8	Enviar señal al proceso.
let	Comando	6	Asignación aritmética de variables.
newgrp	Comando		Iniciar nuevo shell con nuevo ID de grupo (obsoleto).
print	Comando	1	Expande e imprime los argumentos en la salida estándar.
printf	Comando	7	Expande e imprime los argumentos en la salida estándar, utilizando los especificadores de formato *printf(3)* de ANSI C.
pwd	Comando	1	Imprimir directorio de trabajo.
read	Comando	7	Leer una línea de la entrada estándar.
readonly	Comando	5	Hacer que las variables sean de sólo lectura (no asignables).

Nombre	**Comando / Palabra Clave**	**Capítulo**	**Resumen**
return	Comando	5	Retorno desde función o script circundante.
select	Palabra clave	5	Constructo de generación de menús.
set	Comando	3	Configura las opciones.
shift	Comando	6	Cambia los argumentos de la línea de comandos.
sleep	Comando	8	Suspende la ejecución durante el número de segundos indicado.
test	Comando	5	Versión antigua del programa de prueba condicional. Utilice `[[...]]` en su lugar.
time	Palabra clave		Ejecuta el comando e imprime los tiempos de ejecución. Por sí mismo, imprime los tiempos acumulados para el shell y todos los hijos.
trap	Comando	8	Establezca una rutina de captación de señales.
true	Comando	8	No hacer nada y salir con éxito. Útil para hacer bucles infinitos.
typeset	Comando	6	Establecer características especiales de variables y funciones.
ulimit	Comando	10	Establecer/mostrar los límites de recursos del proceso.
umask	Comando	10	Establecer/mostrar máscara de permisos de archivo.
unalias	Comando	3	Eliminar definiciones de alias.
unset	Comando	3	Eliminar definiciones de variables o funciones.
until	Palabra clave	5	Construcción en bucle.
wait	Comando	8	Espere a que termine(n) la(s) tarea(s) en segundo plano.
whence	Comando	3	Identificar la fuente de mando.

Nombre	Comando / Palabra Clave	Capítulo	Resumen
while	Palabra clave	5	Construcción en bucle.

Las asignaciones para los comandos *alias, export, readonly* y *typeset* se procesan como asignaciones de variables, en el sentido de que la expansión de tilde se realiza después del carácter =, y la división de campos no se realiza en ninguna sustitución de variable en el valor que se asigna.

B.3. Alias predefinidos

Hay una serie de alias predefinidos, es decir, incorporados automáticamente a *ksh* en tiempo de compilación. Están listados en la siguiente tabla. Tenga en cuenta que algunos de ellos se definen con un carácter de espacio al final. Esto permite la expansión del alias en la palabra que sigue al alias en la línea de comandos.

Nombre	Capítulo	Valor completo
autoload	4, 6	`alias autoload='typeset -fu'`
command	7	`alias command='command '`
fc	2	`alias fc=hist`
float	6	`alias float='typeset -E'`
functions	6	`alias functions='typeset -f'`
hash	3	`alias hash='alias -t --'`
history	2	`alias history='hist -l'`
integer	6	`alias integer='typeset -i'`
nameref	4	`alias nameref='typeset -n'`
nohup	3,8	`alias nohup='nohup '`
r	2	`alias r='hist -s'`
redirect	9	`alias redirect='command exec'`
stop	8	`alias stop='kill -s STOP'`
times		`alias time='{ {time;} 2>&1;}'`
type	4	`alias type='whence -v'`

B.4. Variables de Shell incorporadas

He aquí un resumen de todas las variables de shell incorporadas:

Variable	Capítulo	Significado
#	4	Número de argumentos dados al proceso actual.

Variable	**Capítulo**	**Significado**
0	4	Argumentos de línea de comandos para el proceso actual. Dentro de comillas dobles, se expande a argumentos individuales.
*	4	Argumentos de línea de comandos para el proceso actual. Dentro de comillas dobles, se expande a un único argumento.
- (guión)		Opciones dadas al shell en la invocación.
?	5	Estado de salida del comando anterior.
$	8	ID de proceso del proceso shell.
_ (guón bajo)	3	Dentro de `$MAILPATH`: el nombre de archivo que activó el mensaje «tienes correo». En la línea de comandos: último argumento del comando anterior. Dentro de un script: la ruta completa utilizada para encontrar y ejecutar el script.
!	8	ID de proceso del último comando en segundo plano.
.sh.edchar	10	Caracteres introducidos al procesar una trampa KEYBD.
.sh.edcol	10	Posición del cursor en la trampa KEYBD más reciente.
.sh.edmode	10	Igual a ESC en modo *vi*, vacío en caso contrario.
.sh.edtext	10	Caracteres en el búfer de entrada durante una trampa KEYBD.
.sh.match	4	Variable de matriz con texto que coincide con el patrón en la sustitución de variables. (A partir de *ksh93l*.)
.sh.name	9	Nombre de una variable que ejecuta una función de disciplina.
.sh.subscript	9	Subíndice de una variable de matriz que ejecuta una función de disciplina.
.sh.value	9	Valor de la variable que ejecuta una función de disciplina.
.sh.version	4, 9	Versión de *ksh*.
CDPATH	3	Lista de directorios en los que debe buscar el comando *cd*.

Variable	**Capítulo**	**Significado**
COLUMNS	3	Anchura de visualización en columnas (para los modos de edición y `select`).
EDITOR	2	Se utiliza para establecer el modo de edición; también lo utilizan *mail* y otros programas. Anulado por `VISUAL`, si está configurado.
EMPV	3	Nombre del archivo que se ejecutará como archivo de entorno cuando se invoque al shell.
FCEDIT	2	Editor por defecto obsoleto para el comando *hist*.
FIGNORE	1	Patrón para los archivos a ignorar durante la expansión del patrón.
FPATH	4	Ruta de búsqueda de funciones autocargadas.
HISTCMD	2	Número del comando actual en el historial de comandos.
HISTEDIT	2	Editor por defecto para el comando *hist*
HISTFILE	2	Nombre del archivo de historial de comandos.
HISTSIZE	2	Número de líneas guardadas en el archivo histórico.
HOME	3	Directorio de inicio (login).
IFS	7	Separador de campo interno: lista de caracteres que actúan como separadores de palabras. Normalmente se establece en espacio, TAB y nueva línea.
LANG		Nombre por defecto de la configuración regional actual; reemplazado por otras variables LC_*.
LC_ALL		Nombre de la configuración regional actual; sustituye a LANG y a las demás variables LC_*.
LC_COLLATE		Nombre de la configuración regional actual para la clasificación de caracteres.
LC_CTYPE		Nombre de la configuración regional actual para determinar la clase de caracteres durante la comparación de patrones; véase el Capítulo 4.
LC_NUMERIC		Nombre de la configuración regional actual para el formato de los números (punto decimal, separador de miles).

Variable	**Capítulo**	**Significado**
LINEO	9	Número de línea del script o función que se acaba de ejecutar.
LINES	3	Altura de la pantalla en líneas (para el comando `select`).
MAIL	3	Nombre del archivo para comprobar si hay correo nuevo.
MAILCHECK	3	Frecuencia (en segundos) con la que se comprueba si hay correo nuevo.
MAILPATH	3	Lista de nombres de archivo para comprobar si hay correo nuevo, si MAIL no está configurado.
OLDPWD	3	Directorio de trabajo anterior.
OPTARG	6	Argumento de la opción procesada por *getopts*.
OPTIND	6	Número del primer argumento después de las opciones.
PATH	3	Ruta de búsqueda de comandos.
PPID	8	ID del proceso padre.
PS1	3	String del símbolo del sistema principal.
PS2	3	String de solicitud de continuación de línea.
PS3	5	String de consulta para el comando `select`.
PS4	9	String de consulta para la opción `xtrace`.
PWD	3	Directorio de trabajo actual.
RANDOM	9	Número aleatorio entre 0 y 32767 ($2^15 - 1$).
REPLY	5, 7	Respuesta del usuario al comando `select`; resultado del comando *read* si no se dan nombres de variables.
SECONDS	3	Número de segundos transcurridos desde que se invocó al shell.
SHELL	3	Ruta completa de los programas shell que se deben utilizar para ejecutar comandos.
TMOUT	5, 7, 10	Si se establece en un número entero positivo, número de segundos entre comandos tras los cuales el shell termina automáticamente. También se aplica a la lectura de respuestas a `select` y *read*.
VISUAL	2	Permite establecer el modo de edición.

B.5. Operadores de prueba

Son los operadores que se utilizan con la construcción `[[...]]`. Pueden combinarse lógicamente con `&&` ("y") y `||` ("o") y agruparse con paréntesis. Cuando se utilizan con nombres de archivo de la forma `/dev/fd/N`, comprueban el atributo correspondiente del descriptor de archivo abierto *N*.

Operador	**Verdadero si**
-a file	*file* existe (Obsoleto. `-e` es preferido)
-b file	*file* es un dispositivo de bloque
-c file	*file* es un dispositivo de caracteres
-d file	*file* es un directorio
-e file	*file* existe
-f file	*file* es un archivo regular
-g file	*file* tiene activado el bit de *setgid*
-G file	El ID de grupo de *file* es el mismo que el ID de grupo efectivo del intérprete de comandos
-h file	*file* es un enlace simbólico
-k file	*file* tiene su bit *sticky* activado
-l file	*file* es un enlace simbólico. (Solo funciona en sistemas en los que `/bin/test -l` comprueba si hay enlaces simbólicos
-L file	*file* es un enlace simbólico
-n string	strin no es nulo
-o option	Option está activado
-O file	*file* es propiedad del ID de usuario efectivo del intérprete de comandos
-p file	*file* es una tubería o una tubería con nombre (archivo FIFO)
-r file	*file* tiene permisos de lectura
-s file	*file* no está vacío
-S file	*file* es un socket
-t file	El descriptor de fichero *N* apunta a una terminal
-u file	*file* tiene activado el bit *setuid*
-w file	*file* tiene permisos de escritura

Operador	**Verdadero si**
-x file	*file* tiene permisos de ejecución, o el archivo es un directorio en el que se puede buscar
-z string	*string* es nulo
fileA -nt fileB	fileA es más reciente que file B, o fileB no existe
fileA -ot fileB	fileA es más antiguo que fileB, o fileB no existe
fileA -ef fileB	fileA y fileB apuntan al mismo archivo
string = pattern	*string* coincide con el patrón (que puede contener comodines). Obsoleto; se prefiere ==
string == pattern	*String* coincide con el patrón (que puede contener comodines)
string != pattern	String no coincide con el patrón
stringA < stringB	*StringA* está antes que *StringB* en el orden del diccionario
stringA > stringB	*StringA* va después que *StringB* en el orden del diccionario
exprA -eq expB	Las expresiones aritméticas *exprA* y *exprB* son iguales
exprA -ne exprB	Las expresiones aritméticas *exprA* y *exprB* no son iguales
exprA -lt exprB	*exprA* es menor que *exprB*
exprA -gt exprB	*exprA* es mayor que *exprB*
exprA -le exprB	*exprA* es menor o igual que *exprB*
exprA -ge exprB	*exprA* es mayor o igual que *exprB*

Los operadores *-eq, -ne, -lt, -le, -gt* y *-ge* se consideran obsoletos en *ksh93*; en su lugar debe utilizarse el comando *let* o `((...))`.

Para =, == y !=, cite el patrón para realizar comparaciones literales de strings.

B.6. Opciones

Son opciones que pueden activarse con el comando `set -o`. Todas están inicialmente desactivadas excepto donde se indique lo contrario. Las abreviaturas, cuando aparecen, son opciones de set que pueden utilizarse en lugar de la orden `set -o` completa (por ejemplo, `set -a` es una abreviatura de `set -o allexport`). En la mayoría de los casos, las abreviaturas son opciones del shell de Bourne compatibles con versiones anteriores. Para desactivar una opción, utilice `set +o nombre largo` o `set +X`, donde *nombre largo* y *X* son la «forma larga» o la forma de «un solo carácter» de la opción, respectivamente.

Opción	**Abreviatura**	**Significado**
allexport	-a	Exporta todas las variables definidas posteriormente.
bgnice		Ejecuta todos los trabajos en segundo plano con prioridad reducida (activada por defecto).
emacs		Utilice la edición de línea de comandos estilo Emacs.
errexit	-e	Salir del shell cuando un comando sale con un estado distinto de cero.
gmacs		Utiliza la edición de línea de comandos al estilo de Emacs, pero con un significado ligeramente diferente para CTRL-T (véase el capítulo 2).
ignoreeof		No permitir CTRL-D para salir del shell.
keyword	-k	Ejecutar asignaciones en medio de líneas de comando. (Muy obsoleto.)
markdirs		Añade `/` a todos los nombres de directorio generados a partir de la expansión de comodines.
monitor	-m	Activar el control de trabajos (activado por defecto).
noclobber	-c	No permitir a > la redirección a archivos existentes.
noexec	-n	Leer los comandos y comprobar si hay errores de sintaxis, pero no los ejecuta.
noglob	-f	Desactivar la expansión de comodines.
nolog		Desactivar el historial de comandos para las definiciones de funciones.
notify	-b	Imprime mensajes de finalización de trabajo inmediatamente, en lugar de esperar a la siguiente solicitud.
nounset	-u	Tratar las variables indefinidas como errores, no como nulos.
pipefail		Espera a que se completen todos los trabajos de una cadena. El estado de salida es el del último comando que falló, o cero en caso contrario. (*ksh93g* y posteriores).
privileged	-p	El script se ejecuta en modo *suid*.
trackall	-h	Crea un alias para cada ruta completa encontrada en una búsqueda de comandos. (*ksh93* ignora esta opción; el comportamiento es siempre activado, incluso si esta opción está desactivada).
verbose	-v	Imprime los comandos (textualmente) antes de ejecutarlos.

Opción	Abreviatura	Significado
vi		Utilice la edición de línea de comandos estilo *vi*.
viraw		Utilice el modo *vi* y haga que cada pulsación de tecla tenga efecto inmediatamente. (Esto es necesario en algunos sistemas muy antiguos para que el modo *vi* funcione, y es necesario en todos los sistemas para poder usar TAB para completar. A partir de *ksh93n*, se habilita automáticamente cuando se usa *vi-mode*).
xtrace	-x	Imprime los comandos (después de las expansiones) antes de ejecutarlos.

El comando set tiene algunas opciones adicionales que no tienen sus correspondientes versiones set -o, como se indica a continuación:

Opción	Significado
set -A ...	Asignación de matrices indexadas.
set -s	Ordena los parámetros posicionales.
set -t	Leer y ejecutar un comando, y luego salir. (Obsoleto.)

B.7. Opciones tipográficas

Estas son las opciones del comando *typeset*. Utilice `+option` para desactivar una opción, por ejemplo, `typeset +x foo` para dejar de exportar la variable foo.

Opción	Significado
	Sin opción, crear variable local dentro de la función.
-A	Declarar variable como array asociativo.
-E[*n*]	Declara la variable como un número de punto flotante. Opcional *n* es el número de cifras significativas.
-F[*n*]	Declara la variable como un número de punto flotante. Opcional *n* es el número de dígitos significativos.
-f	Sin argumentos, imprime todas las definiciones de funciones.
-f *fname*	Imprime la definición de la función *fname*.
+f	Imprime todos los nombres de las funciones.
-ft	Activar el modo de rastreo para la(s) función(es) nombrada(s).
+ft	Desactiva el modo de rastreo para la(s) función(es) nombrada(s).

Opción	Significado
-fu	Definir nombre(s) dado(s) como función(es) *autoloaded*.
-fx	Obsoleto; no hace nada en *ksh93*.
-H	Asignación de nombres de archivo de Unix a host para sistemas no Unix.
-i[*n*]	Declara la variable como un entero. Opcional *n* es la base de salida.
-l	Convierte todas las letras a minúsculas.
-L	Justificar a la izquierda y eliminar los espacios a la izquierda.
-n	Declarar variable como *nameref*.
-p	Imprime comandos *tipográficos* (*typeset*) para volver a crear variables con los mismos atributos.
-r	Hacer que la variable sea de sólo lectura.
-R	Justificar a la derecha y eliminar los espacios finales.
-t	Etiquetar la variable. (Obsoleto.)
-u	Convierte todas las letras a mayúsculas.
-ui[*n*]	Declara la variable como un entero sin signo. Opcional *n* es la base de salida. (*ksh93m* y posteriores).
-x	Variable de exportación, es decir, poner en el entorno para que se pase a los subprocesos.
-Z[*n*]	Justificar a la derecha y rellenar con ceros a la izquierda. *n* es la anchura, o la anchura se establece a partir del valor utilizado en la primera asignación.

B.8. Aritmética

A partir de *ksh93m*, la instalación de aritmética incorporada comprende un gran porcentaje de las expresiones del lenguaje C. Esto hace que el shell sea más atractivo como un lenguaje de programación completo. Las siguientes características están disponibles:

Sufijos de tipo al final

Las constantes enteras pueden tener un sufijo `U` o `L` al final para indicar que son sin signo o largas, respectivamente. Aunque las versiones en minúsculas también se pueden usar, esto no se recomienda, ya que es fácil confundir una `l` (ele minúscula) con un 1 (uno).

Constantes de caracteres en C

Se reconocen las constantes de caracteres individuales en C, entre comillas simples. Al igual que en C, actúan como constantes enteras. Por ejemplo:

```
$ typeset -i c
$ for ((c = 'a'; c <= 'z'; c++))
> do print $c
> done
97
98
99
100
...
```

Constantes octales y hexadecimales

Puedes utilizar el formato C para constantes octales (base 8) y hexadecimales (base 16). Las constantes octales comienzan con un 0 líder, y las constantes hexadecimales comienzan con un `0x` o `0X` líder. Por ejemplo:

```
$ print $((010 + 1))          # Octal 10 es decimal 8
9
$ print $((0x10 + 1))         # Hexadecimal 10 es decimal 16
17
```

Aritmética con enteros sin signo

Al usar `typeset -ui`, puedes crear enteros sin signo. Los enteros regulares representan números positivos y negativos. Los enteros sin signo comienzan en 0, llegan hasta algún valor dependiente de la implementación y luego «se envuelven» nuevamente a 0. Del mismo modo, restar 1 de 0 se envuelve en la otra dirección, dando como resultado el número sin signo más grande:

```
$ typeset -ui u=0
$ let u--
$ print $u
4294967295
```

Operadores y precedencia en C

ksh admite el conjunto completo de operadores en C, con la misma precedencia y asociatividad. Los operadores se presentaron en detalle en el Capítulo 6 y se resumen nuevamente a continuación.

Operador	**Significado**	**Asociatividad**
`++ --`	Incremento y decremento, prefijo y postfijo	De izquierda a derecha

Operador	Significado	Asociatividad
`+ - ! ~`	Más y menos unarios; negación lógica y por bits	De derecha a izquierda
`**`	Exponenciación [1]	De derecha a izquierda
`* ? /%`	Multiplicación, división y resta	De izquierda a derecha
`+ -`	Suma y resta	De izquierda a derecha
`<< >>`	Desplazamiento de bits a izquierda y derecha	De izquierda a derecha
`< <= > >=`	Comparaciones	De izquierda a derecha
`== !=`	Iguales y no iguales	De izquierda a derecha
`&`	Bit a bit «y»	De izquierda a derecha
`^`	«O» exclusivo bit a bit	de Izquierda a derecha
`|`	Bit a bit «o»	De izquierda a derecha
`&&`	AND lógico	De izquierda a derecha
`||`	OR lógico	De izquierda a derecha
`?:`	Expresión condicional	De derecha a izquierda
`= += -= *= /=%= &= ^= <<= >>=`	Operadores de asignación	De derecha a izquierda
`,`	Evaluación secuencial	De izquierda a derecha

B.9. Comandos del modo Emacs

Aquí tienes una lista completa de todos los comandos del modo de edición de *emacs*. Algunos de estos, como `ESC [ A`, representan secuencias de teclas de flecha de terminal estándar ANSI; se agregaron para *ksh93h*.

Comando	Significado
CTRL-A	Mover al principio de la línea
CTRL-B	Retroceder un carácter sin borrar
CTRL-C	Capitalizar el carácter después del punto
CTRL-D	Borrar un carácter hacia adelante
CTRL-E	Mover al final de la línea
CTRL-F	Avanzar un carácter

Comando	**Significado**
CTRL-I (TAB)	Completar el nombre de archivo en la palabra actual (a partir de *ksh93h*)
CTRL-J	Igual que ENTER.
CTRL-K	Borrar («kill») hacia adelante hasta el final de la línea
CTRL-L	Limpiar pantalla
CTRL-M	Igual que ENTER
CTRL-N	Línea siguiente
CTRL-O	Igual que ENTER, luego muestra la línea siguiente en el archivo de historial
CTRL-P	Línea anterior
CTRL-R	Buscar hacia atrás
CTRL-T	Intercambiar los dos caracteres a cada lado del punto
CTRL-U	Repetir el siguiente comando cuatro veces
CTRL-V	Imprimir la versión del shell de Korn
CTRL-W	Borrar («wipe») todos los caracteres entre el punto y la marca
CTRL-X CTRL-E	Invocar el programa *emacs* en el comando actual
CTRL-X CTRL-X	Intercambiar el punto y la marca
CTRL-Y	Recuperar («yank») el último elemento eliminado
CTRL-] x	Buscar hacia adelante x, donde x es cualquier carácter
CTRL-@	Establecer la marca en el punto
DEL	Borrar un carácter hacia atrás
CTRL-[	Igual que ESC (en la mayoría de los teclados)
ESC b	Mover una palabra hacia atrás
ESC c	Cambiar la palabra después del punto a mayúsculas
ESC d	Borrar una palabra hacia adelante
ESC f	Mover una palabra hacia adelante
ESC h	Borrar una palabra hacia atrás
ESC l	Cambiar la palabra después del punto a minúsculas
ESC p	Guardar caracteres entre el punto y la marca como si estuvieran borrados

Comando	**Significado**
ESC CTRL-H	Borrar una palabra hacia atrás
ESC CTRL-] x	Buscar hacia atrás x, donde x es cualquier carácter
ESC ESPACIO	Establecer la marca en el punto
ESC #	Insertar línea en el archivo de historial para edición futura
ESC DEL	Borrar una palabra hacia atrás
ESC <	Mover al principio del archivo de historial
ESC >	Mover al final del archivo de historial
ESC .	Insertar última palabra en la línea de comando anterior después del punto
ESC _	Igual que arriba
ESC ESC	Hacer la completación de nombre de archivo/comando/variable en la palabra actual
ESC *	Hacer la expansión de nombre de archivo/comando/variable en la palabra actual
ESC =	Hacer la lista de nombre de archivo/comando/variable en la palabra actual
ESC [A	Línea anterior (*ksh93h* y más reciente)
ESC [B	Línea siguiente (*ksh93h* y más reciente)
ESC [C	Mover un carácter hacia adelante (*ksh93h* y más reciente)
ESC [D	Mover un carácter hacia atrás (sin borrar) (*ksh93h* y más reciente)
ESC [H	Mover al principio de la línea (*ksh93h* y más reciente)
ESC [Y	Mover al final de la línea (*ksh93h* y más reciente)
Kill	El carácter de eliminación *stty(1)*, a menudo CTRL-U o @ o CTRL-X. Esto borra todo en la línea. Escribirlo dos veces activa el modo «line feed», que emite un carácter de avance de línea para empezar en una nueva línea. Esto es apropiado para terminales solo de papel.

B.10. Comandos del modo de control vi

Aquí tienes una lista completa de todos los comandos de control en el modo *vi*. Al igual que con los comandos del modo *emacs*, las secuencias como `[ A` son para las teclas de flecha estándar ANSI y se agregaron para *ksh93h*.

Comando	**Significado**
h	Mover a la izquierda un carácter
[	D Mover a la izquierda un carácter (*ksh93h* y más reciente)
l	Mover a la derecha un carácter
space	Mover a la derecha un carácter
[C	Mover a la derecha un carácter (*ksh93h* y más reciente)
w	Mover a la derecha una palabra
b	Mover a la izquierda una palabra
W	Mover al principio de la siguiente palabra no en blanco
B	Mover al principio de la palabra no en blanco precedente
e	Mover al final de la palabra actual
E	Mover al final de la palabra actual no en blanco
0	Mover al principio de la línea
[H	Mover al principio de la línea (*ksh93h* y más reciente)
^	Mover al primer carácter no en blanco en la línea
$	Mover al final de la línea
[Y	Mover al final de la línea (*ksh93h* y más reciente)
i	Insertar texto antes del carácter actual
a	Insertar texto después del carácter actual
I	Insertar texto al principio de la línea
A	Insertar texto al final de la línea
r	Reemplazar un carácter (no entra en modo de entrada)
R	Sobrescribir el texto existente
dh	Eliminar un carácter hacia atrás
dl	Eliminar un carácter hacia adelante
db	Eliminar una palabra hacia atrás
dw	Eliminar una palabra hacia adelante
dB	Eliminar una palabra no en blanco hacia atrás
dW	Eliminar una palabra no en blanco hacia adelante
d$	Eliminar hasta el final de la línea
d0	Eliminar hasta el principio de la línea
D	Equivalente a d$ (eliminar hasta el final de la línea)
dd	Equivalente a 0d$ (eliminar toda la línea)

Comando	**Significado**
C	Equivalente a c$ (eliminar hasta el final de la línea, entrar en modo de entrada)
cc	Equivalente a 0$ (eliminar toda la línea, entrar en modo de entrada)
s	Equivalente a xi (eliminar el carácter actual, entrar en modo de entrada)
S	Equivalente a cc (eliminar toda la línea, entrar en modo de entrada)
x	Equivalente a dl (eliminar carácter hacia atrás)
X	Equivalente a dh (eliminar carácter hacia adelante)
k o -	Mover hacia atrás una línea [A Mover hacia atrás una línea (*ksh93h* y más reciente)
j o +	Mover hacia adelante una línea
[B	Mover hacia adelante una línea (*ksh93h* y más reciente)
G	Mover a la línea dada por el recuento de repeticiones
/string	Buscar hacia adelante la cadena
?string	Buscar hacia atrás la cadena
n	Repetir la búsqueda hacia adelante
N	Repetir la búsqueda hacia atrás
f x	Mover a la siguiente ocurrencia de x hacia la derecha
F x	Mover a la ocurrencia anterior de x hacia la izquierda
t x	Mover a la siguiente ocurrencia de x hacia la derecha, luego hacia atrás un espacio
T x	Mover a la ocurrencia anterior de x hacia la izquierda, luego hacia adelante un espacio
yh	Yankar un carácter hacia atrás
yl	Yankar un carácter hacia adelante
yb	Yankar una palabra hacia atrás
yw	Yankar una palabra hacia adelante
yB	Yankar una palabra no en blanco hacia atrás
yW	Yankar una palabra no en blanco hacia adelante
y$	Yankar hasta el final de la línea
y0	Yankar hasta el principio de la línea
Y	Equivalente a y$ (yankar hasta el final de la línea)

Comando	**Significado**
yy	Equivalente a 0y$ (yankar toda la línea)
u	Deshacer el último cambio de edición
U	Deshacer todos los cambios de edición realizados en la línea
. (punto)	Repetir el último comando de edición
|	Mover a la posición absoluta de la columna
;	Rehacer el último comando de búsqueda de caracteres
,	Rehacer el último comando de búsqueda de caracteres, pero en dirección opuesta
%	Mover a la coincidencia de (,), {, }, [, o]
\	Realizar la completación de nombre de archivo/comando/variable
CTRL-I (TAB)	Realizar la completación de nombre de archivo/comando/variable (solo para `set -o viraw`) (a partir de *ksh93h*)
*	Realizar la expansión de nombre de archivo/comando/variable (en la línea de comandos)
=	Realizar la expansión de nombre de archivo/comando/variable (como lista impresa)
~	Invertir («twiddle») mayúsculas y minúsculas del carácter actual
_	Anexar la última palabra del comando anterior, entrar en modo de entrada
v	Ejecutar el comando hist en la línea actual (en realidad, ejecutar el comando `hist -e ${VISUAL:-${EDITOR:-vi}}`); generalmente esto significa ejecutar *vi* completo en la línea actual
CTRL-J	Igual que ENTER
CTRL-L	Iniciar una nueva línea y volver a dibujar la línea actual en ella
CTRL-M	Igual que ENTER
CTRL-V	Imprimir la versión del shell de Korn
#	Agregar # (carácter de comentario) a la línea y enviarlo. Si la línea comienza con #, quitar el # inicial y todos los # iniciales después de cualquier salto de línea incrustado
@ x	Insertar expansión de alias _x como entrada de modo de comando

B.11. Uso de *getopts*

El comando *getopts* es extremadamente capaz. Con él, puedes hacer que tus scripts de shell acepten opciones largas, especificar que los argumentos son opcionales o numéricos, y proporcionar descripciones de los argumentos y valores para que las opciones `-?`, `--man`, `--html` y `--nroff` funcionen de la misma manera para tu programa que para los comandos internos de *ksh93*.

El precio por este poder es la complejidad del «lenguaje» de descripción de opciones. Basándonos en una descripción proporcionada por el Dr. Glenn Fowler de AT&T Research, describimos cómo evolucionaron las facilidades, cómo funcionan y resumimos cómo usarlas en tus propios programas. Usamos el comando *getopts* extendido en la solución para la Tarea B-1.

Tarea B-1

Diseña el programa *phaser4*, que combina las características de los programas *phaser3* y *tricorder*. Asegúrate de que sea autosuficiente.[a]

[a] No, las paredes de mi habitación no están cubiertas de pósters de Star Trek. Superé eso hace mucho tiempo, y además, mi esposa no me dejaría de todos modos. ADR.

El primer paso es describir las opciones. Esto se hace con un comentario en la parte superior del script:

```
# uso: phaser4 [ opciones ] archivos
# -k, --kill                  usar configuración de eliminación (predeterminado)
# -l n, --level n             establecer el nivel del phaser (predeterminado = 2)
# -s, --stun                  usar configuración solo de aturdimiento
# -t [lf], --tricorder [lf]   modo tricorder, escaneo opcional para forma de vida lf
```

Ahora comienza la diversión. Este esquema de capacidades sigue el orden en que se agregaron funciones a *getopts*.

1. Comienza con el comando *getopts* como se describe en el Capítulo 6. Esto produce una cadena de opciones simple que solo permite opciones de una sola letra:

   ```
   USAGE="kl#st:"
   while getopts "$USAGE" optchar ...
   ```

2. Añade una descripción textual para el argumento de opción. Esto se hace encerrando texto arbitrario entre [y]:

   ```
   USAGE="kl#[level]st:[life_form]"
   while getopts "$USAGE" optchar ...
   ```

3. Permite un valor predeterminado para el argumento de una opción. Esto se hace especificando `:= value` dentro de la descripción entre corchetes:

```
USAGE="kl#[level:=2]st:[life_form]"
while getopts "$USAGE" optchar ...
```

4. Añade ? después de : para indicar un argumento opcional:

```
USAGE="kl#[level:=2]st:?[life_form]"
while getopts "$USAGE" optchar ...
```

5. Permite opciones largas que comienzan con `--`. Esto se hace usando `[let:long]` en lugar de la única letra de opción:

```
USAGE="[k:kill]"
USAGE+="[l:level]#[level:=2]"
USAGE+="[s:stun]"
USAGE+="[t:tricorder]:?[life_form]"
while getopts "$USAGE" optchar ...
```

   Aquí, hemos dividido cada opción en su propia línea para facilitar el seguimiento y las hemos concatenado usando el operador de asignación +=. Ten en cuenta que no hay saltos de línea en la cadena.

6. Dentro de los corchetes cuadrados de una letra de opción, permite que siga un texto descriptivo después de un signo de interrogación. Este texto se ignora, al igual que cualquier carácter de espacio en blanco, incluidos los saltos de línea:

```
USAGE="[k:kill?Usar configuración de eliminación (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]"
USAGE+="[s:stun?¡Solo aturdimiento!]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
while getopts "$USAGE" optchar ...
```

7. Ahora se pone interesante. Los encabezados de sección al estilo de la página man de Unix vienen *antes* de la descripción de la opción. Se distinguen de las descripciones de las opciones al comenzar con un + dentro de corchetes cuadrados:

```
USAGE="[+NOMBRE?phaser4 --- phaser y tricorder combinados]"
USAGE+="[+DESCRIPCIÓN?El programa phaser4 combina la operación "
USAGE+="de los programas phaser3 y tricorder en una herramienta práctica.]"
USAGE+="[k:kill?Usar configuración de eliminación (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]"
USAGE+="[s:stun?¡Solo aturdimiento!]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
while getopts "$USAGE" optchar ...
```

   Observa que *getopts* comprende automáticamente que la descripción real de las opciones viene después de los encabezados de la página man; no hay un explícito

`[+OPTIONS?...]` en el texto del string.

8. Se puede proporcionar texto adicional descriptivo para el resumen de uso corto después de la descripción de las opciones, separado por dos saltos de línea:

```
USAGE="[+NOMBRE?phaser4 --- phaser y tricorder combinados]"
USAGE+="[+DESCRIPCIÓN?El programa phaser4 combina la operación "
USAGE+="de los programas phaser3 y tricorder en una herramienta práctica.]"
USAGE+="[k:kill?Usar configuración de eliminación (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]"
USAGE+="[s:stun?¡Solo aturdimiento!]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
USAGE+=$'\n\narchivo ...\n\n'   # Usa cadena ANSI C para el carácter \n
USAGE+="[+VER TAMBIÉN?phaser3(1), tricorder(1)]"
while getopts "$USAGE" optchar ...
```

9. Para indicar texto que debe cursivarse, enciérralo entre pares de caracteres \a. Para indicar texto que debe ponerse en negrita, enciérralo entre pares de caracteres \b:

```
USAGE="[+NOMBRE?phaser4 --- phaser y tricorder combinados]"
USAGE+="[+DESCRIPCIÓN?El programa \aphaser4\a combina la operación "
USAGE+="de los programas \aphaser3\a y \atricorder\a en una herramienta práctica.]"
USAGE+="[k:kill?Usar configuración de eliminación (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]"
USAGE+="[s:stun?¡Solo aturdimiento!]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
USAGE+=$'\n\narchivo ...\n\n'   # Usa cadena ANSI C para el carácter \n
USAGE+=$'[+VER TAMBIÉN?\aphaser3\a(1), \atricorder\a(1)]'
while getopts "$USAGE" optchar ...
```

10. Es posible un control dinámico de la salida descriptiva. Para hacer esto, escribe una función que imprima lo que desees y luego encierra el nombre de la función entre un par de caracteres \f: \fname\f (esto no es necesario para phaser4).

11. Si una opción (o cualquier otra cosa) necesita una descripción detallada, encerrar el texto entre y crea una lista sangrada. Esto es particularmente útil para describir diferentes valores de opción:

```
USAGE="[+NOMBRE?phaser4 --- phaser y tricorder combinados]"
USAGE+="[+DESCRIPCIÓN?El programa \aphaser4\a combina la operación "
USAGE+="de los programas \aphaser3\a y \atricorder\a en una herramienta práctica.]"
USAGE+="[k:kill?Usar configuración de eliminación (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]{ Añadir descripciones de
    valores
                [level=0-2?configuraciones no letales]
                [level=3-10?letales, usar con precaución]
}"
USAGE+="[s:stun?¡Solo aturdimiento!]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
USAGE+=$'\n\narchivo ...\n\n'   # Usa cadena ANSI C para el carácter \n
```

```
USAGE+=$'[+VER TAMBIÉN?\aphaser3\a(1), \atricorder\a(1)]'
while getopts "$USAGE" optchar ...
```

12. Casi terminado. El texto entre corchetes cuadrados que comienza con un signo menos proporciona información de versión e identificación. Tal texto va al principio. El ítem vacío indica una versión y puede contener tanto cadenas de SCCS como de RCS, como se muestra aquí:

```
USAGE=$'[-?\n@(#)$Id: phaser4 (Investigación y Desarrollo de la Flota Estelar)'
USAGE+=$' Stardate 57234.22 $\n]'
USAGE+="[-autor?J. Programador ]"
USAGE+="[-derechos?Copyright (c) Stardate 57000 Flota Estelar.]"
USAGE+="[-licencia?http://www.starfleet.mil.fed/weapons-license.xml23]"
USAGE+="[+NOMBRE?phaser4 --- phaser y tricorder combinados]"
USAGE+="[+DESCRIPCIÓN?El programa \aphaser4\a combina la operación "
USAGE+="de los programas \aphaser3\a y \atricorder\a en una herramienta práctica.]"
USAGE+="[k:kill?Usar configuración de eliminación (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]{
                [level=0-2?configuraciones no letales]
                [level=3-10?letales, usar con precaución]
}"
USAGE+="[s:stun?¡Solo aturdimiento!]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
USAGE+=$'\n\narchivo ...\n\n'   # Usa cadena ANSI C para el carácter \n
USAGE+=$'[+VER TAMBIÉN?\aphaser3\a(1), \atricorder\a(1)]'
while getopts "$USAGE" optchar ...
```

13. Finalmente, permite escapes dentro de las cadenas. `]]` representa un corchete de cierre literal cuando *getopts* de otra manera podría interpretarlo como un corchete de cierre. De manera similar, ?? representa un signo de interrogación literal que de otra manera podría iniciar una descripción.

Uff, ¡eso es mucha información! Sin embargo, verlo en el orden en que se agregó ayuda a que tenga sentido. Aquí tienes un resumen de los elementos que van en la cadena de uso, en el orden que requiere *getopts*:

1. Las cadenas de identificación para la versión, autor, licencia, etc., son la primera parte. Están encerradas en corchetes y comienzan con un signo menos. El nombre del ítem, como «author», sigue el signo menos y termina en un signo de interrogación. Después del signo de interrogación viene la información asociada.

   El ítem vacío indica información de versión y debe tener la forma mostrada anteriormente; *getopts* elimina los caracteres de identificación especiales de SCCS y RCS.

2. Las secciones y el texto de estilo de la página de manual de Unix vienen a continua-

ción. Están encerrados en corchetes y comienzan con un signo más. El nombre de la sección termina en el carácter ?, y el texto descriptivo sigue.

El texto separado por dos nuevas líneas desde la descripción de las opciones se añade al mensaje breve de uso.

3. Las descripciones de las opciones forman la tercera sección. Todavía se permite la forma corta original, como se describió en el Capítulo 6:
   - Usa : para opciones que requieren argumentos.
   - Usa # para opciones que requieren argumentos numéricos.
   - Usa :? y #? para opciones que permiten argumentos pero no los requieren.
4. Sigue las opciones con un texto descriptivo entre [y]. Usa := dentro del texto descriptivo para especificar un valor predeterminado para un argumento de opción.
5. Las opciones largas se emparejan con una letra de opción corta encerrándolas entre corchetes y separándolas por dos puntos. Esto reemplaza la forma de una sola letra.
6. Encierra entre dos caracteres `\a` los elementos que se deben cursivar. Encierra entre dos caracteres `\` los elementos que se deben poner en negrita. Encierra el nombre de una función de personalización a llamar entre dos caracteres `\f`.
7. Usa { y } para encerrar descripciones de opciones anidadas e indentadas.
8. Sigue la sección de opciones con dos nuevas líneas y texto adicional para el resumen de las opciones cortas.
9. Usa `]]` para representar un corchete de cierre literal y `??` para representar un signo de interrogación literal.

Aquí está la versión esquelética de *phaser4*:

```
#! /bin/ksh

# uso: phaser4 [ opciones ] archivos
# -k, --kill                        usa la configuración de matar (predeterminado)
# -l n, --level n                   establece el nivel del phaser (predeterminado = 2)
# -s, --stun                        usa la configuración de aturdir
# -t [lf], --tricorder [lf]         modo tricorder, escaneo opcional para la forma de vida lf

USAGE=$'[-?\n@(#)$Id: phaser4 (Investigación y Desarrollo de la Flota Estelar)'
USAGE+=$' Fecha estelar 57234.22 $\n]'
USAGE+="[-autor?J. Programador <J.Prog@r-d.starfleet.mil.fed>]"
USAGE+="[-copyright=Derechos de autor (c) Fecha estelar 57000 Flota Estelar.]"
```

```
USAGE+="[-licencia?http://www.flotaestelar.mil.fed/licencia-armas.xml23]"
USAGE+="[+NOMBRE?phaser4 --- combinación de phaser y tricorder]"
USAGE+="[+DESCRIPCIÓN?El programa \aphaser4\a combina la operación "
USAGE+="de los programas \aphaser3\a y \atricorder\a en una herramienta práctica.]"
USAGE+="[k:kill?Usar configuración de matar (predeterminado).]"
USAGE+="[l:level]#[level:=2?Establecer el nivel del phaser.]{
              [0-2?configuraciones no letales]
              [3-10?letales, usar con precaución]
}"
USAGE+="[s:stun?Solo aturdir.]"
USAGE+="[t:tricorder?Modo tricorder.]:?[life_form]"
USAGE+=$'\n\narchivo ...\n\n'
USAGE+=$'[+VER TAMBIÉN?\aphaser3\a(1), \atricorder\a(1)]'

kill=1 aturdir=0 nivel=2             # valores predeterminados
tricorder=0 phaser=1
life_form=
while getopts "$USAGE" optchar ; do
  case $optchar in
    k) kill=1 aturdir=0 ;;
    s) kill=0 aturdir=1 ;;
    l) level=$OPTARG
       if ((level < 0)) ; then
         level=0 ;
       fi
       if ((level > 10)) ; then
         level=10 ;
       fi
       ;;
    t) phaser=0 tricorder=1
       life_form=${OPTARG:-"desconocido_general"}
       ;;
  esac
done

print kill=$kill
print aturdir=$aturdir
print level=$level
print phaser=$phaser
print tricorder=$tricorder
print life_form=$life_form
```

Aquí está la salida de `phaser4 --man`:

```
NAME
    phaser4 --- combinación de phaser y tricorder

SYNOPSIS
    phaser4 [ opciones ] archivo ...

DESCRIPTION
```

```
    El programa phaser4 combina la operación de los programas phaser3 y tricorder
    en una herramienta práctica.

OPTIONS
    -k, --kill          Usar configuración de kill (predeterminado).
    -l, --level=level   Establecer el level del phaser.
                           level=0-2
                                 configuraciones no letales
                           level=3-10
                                 letales, usar con precaución
                        El valor predeterminado es 2.
-s, --stun              Solo aturdir.
-t, --tricorder[=life_form]
                        Modo tricorder. El valor de la opción puede omitirse.

SEE ALSO
    phaser3(1), tricorder(1)

IMPLEMENTATION
    version             phaser4 (Investigación y Desarrollo de la Flota Estelar) Fecha estelar
                        57234.22
    autor               J. Programador
    copyright           Copyright (c) Fecha estelar 57000 Flota Estelar.
    license             http://www.flotaestelar.mil.fed/licencia-armas.xml23
```

CAPÍTULO C

CONSTRUCCIÓN DE KSH A PARTIR DEL CÓDIGO FUENTE

Este apéndice describe cómo descargar binarios para *ksh93*, así como cómo descargar el código fuente para *ksh93* y construir una versión funcional. Deberías hacer esto si tu sistema no tiene *ksh93* en absoluto o si necesitas alguna de las características que solo están disponibles en las versiones más recientes.

C.1. Sitios web de Korn Shell

El punto de partida para todo lo relacionado con el shell de Korn es http://www.kornshell.com, mantenido por David Korn, con enlaces agrupados en las siguientes categorías:

Información Al hacer clic en este enlace, se accede a una descripción general de una página sobre el shell de Korn.

Software Al hacer clic en este enlace, se encuentran enlaces al sitio de descargas de AT&T (ver la siguiente sección), algún código de ejemplo, un artículo en línea para *dtksh* y un enlace para *tksh*. Estos dos últimos se describen en el Apéndice A.

Documentación Este enlace lleva a una página con enlaces a información en línea, incluida información general, páginas de manual tanto para *ksh88* como para *ksh93*, libros y referencias sobre el shell de Korn, y documentos sobre el shell de Korn de diversas conferencias.

Recursos Una lista de enlaces a otros recursos en la web para *ksh* y muchos de los otros shells descritos en el Apéndice A, como *bash* y *dtksh*.

Diversión David G. Korn, el programador, se encuentra con KoRN, el grupo de rock. Dicho y hecho.

C.2. Lo que puedes descargar

http://www.research.att.com/sw/download es el punto de partida para descargar el software de *ksh*. El software está cubierto por una licencia de estilo de código abierto. La versión actual de la licencia se encuentra en http://www.research.att.com/sw/license/ast-open.html. Esta licencia se reproduce en el Apéndice D. Deberías leerla y entenderla primero; si sus términos no son aceptables para ti, no deberías descargar el código fuente o binarios del software desde el sitio web de AT&T.

El software en el sitio web de AT&T está disponible en diferentes «paquetes», la mayoría de los cuales tienen nombres con el prefijo «ast», que significa *Advanced Software Tools* (Herramientas de Software Avanzado). Los paquetes de origen vienen como archivos *tar* comprimidos con *gzip*, usando el sufijo de nombre de archivo *.tgz*. Elige uno o más de los siguientes paquetes para descargar:

ratz Un programa ejecutable independiente para leer archivos *tar* comprimidos con *gzip*. Úsalo si no tienes *gzip* en tu sistema y no quieres molestarte en descargar y compilar *gzip* primero. Puedes descargar el código fuente para este paquete o un ejecutable binario para cualquiera de las arquitecturas enumeradas en la Tabla C.1.

ksh Esta es la forma más rápida de obtener un ejecutable de *ksh93*. Hay versiones disponibles para las arquitecturas enumeradas en la Tabla C.1.

INIT Este paquete debe descargarse al construir cualquiera de los siguientes paquetes fuente. Contiene los archivos y estructuras de directorio en los que dependen las herramientas y el sistema de construcción de AST.

ast-ksh Este paquete construye solo la infraestructura de soporte (bibliotecas, programas de prueba de entorno, etc.) para *ksh* y el ejecutable *ksh*. Es lo más sencillo de construir.

ast-base Este paquete construye todo en el paquete *ast-ksh* y algunas herramientas AST básicas adicionales. En particular, incluye *pax*, una herramienta de archivado que combina funciones de *tar(1)* y *cpio(1)*, y *nmake*, una versión significativamente mejorada del programa estándar *make(1)* de Unix. También incluye las bibliotecas *sfio* (Safe Fast I/O) y *ast*, que puedes usar para tus propios programas.

ast-open Este paquete construye todo en el paquete ast-base y muchas herramientas adicionales. Ten en cuenta que *tksh* (ver Apéndice A) está incluido en este paquete.

Cada uno de los paquetes (excepto *INIT*) también está disponible como binarios precompilados. La Tabla C-1 enumera las arquitecturas disponibles para estos paquetes. También hay traducciones de locales para algunos locales y algunos programas.

Tabla C.1: Arquitecturas admitidas para programas AST

Nombre	**SO/Arquitectura**
darwin.ppc	Apple's MacOS X (también conocido como Darwin) para Motorola Power PC
hp.pa	Hewlett-Packard HP-UX para HP Precision Architecture
ibm.risc	AIX de IBM para RS/6000
linux.i386	GNU/Linux en Intel 80386 y superior
linux.s390	GNU/Linux en el mainframe IBM S/390
mvs.390	MVS de IBM en el mainframe IBM S/390
netbsd.i386	NetBSD en Intel 80386 y superior (ver http://www.netbsd.org)
openbsd.i386	OpenBSD en Intel 80386 y superior (ver http://www.openbsd.org)
osf.alpha	OSF/1 en el procesador Compaq (anteriormente Digital) Alpha
sgi.mips3	Silicon Graphics (SGI) Irix en el procesador MIPS
sol.sun4	Solaris 5.4 en la arquitectura Sun SPARC
sol6.sun4	Solaris 5.6 en la arquitectura Sun SPARC
sol7.i386	Solaris 7 en Intel 80386 y superior
sol7.sun4	Solaris 7 en la arquitectura Sun SPARC
sol8.sun4	Solaris 8 en la arquitectura Sun SPARC
sun4	SunOS 4.x en la arquitectura Sun SPARC
unixware.i386	UnixWare (la última versión oficial de System V) en Intel 80386 y superior

C.3. Construcción de ksh (Shell Korn)

Construir cualquiera de los paquetes a partir del código fuente es bastante sencillo. Todos los detalles, junto con preguntas frecuentes y notas, se encuentran en el sitio web de AT&T. Aquí tienes una guía paso a paso de los pasos. Mostramos los pasos para el paquete *ast-open*, pero son idénticos para los demás paquetes de código fuente.

1. Asegúrate de tener un compilador de C para tu sistema. Se prefiere un compilador de C ANSI/ISO, pero un compilador K&R también funcionará. Obtener un compilador de C si no tienes uno está más allá del alcance de este libro; contacta a tu administrador de sistema local.

2. Descarga el/los paquete(s) que deseas construir en un directorio vacío. Aquí, construimos el paquete ast-open del 31 de octubre de 2001:

```
$ ls
INIT.2001-10-31.tgz ast-open.2001-10-31.tgz
```

[0] *ksh93m* y versiones más recientes. El operador ** no está en el lenguaje C.

3. Crea el directorio `lib/package/tgz` y mueve los archivos allí:

```
$ mkdir lib lib/package lib/package/tgz
$ mv *.tgz lib/package/tgz
```

4. Extrae el paquete *INIT* manualmente:

```
$ gzip -d < lib/package/tgz/INIT.2001-10-31.tgz | tar -xvpf -
\r\v\vNOTICE -- LICENSED SOFTWARE -- SEE README FOR DETAILS\r\v\v
README
src/Makefile
src/cmd/Makefile
src/lib/Makefile
...
```

   Si no tienes *gzip*, usa el programa *ratz*, como se describió anteriormente.

5. Inicializa la lista de paquetes disponibles:

```
$ bin/package read
\r\v\vNOTICE -- LICENSED SOFTWARE -- SEE README FOR DETAILS\r\v\v
README
src/Makefile
src/cmd/Makefile
src/lib/Makefile
src/Mamfile
...
```

6. Inicia la compilación. Este paso es bastante detallado y llevará un tiempo. La duración exacta depende de la velocidad de tu sistema y compilador, y del paquete que estás construyendo:

```
$ bin/package make
package: initialize the /home/arnold/ast-open/arch/linux.i386 view
package: update /home/arnold/ast-open/arch/linux.i386/bin/proto
package: update /home/arnold/ast-open/arch/linux.i386/bin/mamake
package: update /home/arnold/ast-open/arch/linux.i386/bin/ratz
package: update /home/arnold/ast-open/arch/linux.i386/bin/release
...
```

7. Instala los archivos creados. Esto se puede hacer con el comando `bin/package install directory`, donde *directory* es la ubicación para colocar las cosas.

   Alternativamente, si todo lo que te interesa es el binario de *ksh*, simplemente puedes copiarlo. El binario compilado estará en un directorio llamado `arch/ARCH/bin`, donde *ARCH* representa tu arquitectura, como *linux.i386*:

```
cp arch/linux.i386/bin/ksh $HOME/bin/ksh93
```

8. ¡Disfruta!

CAPÍTULO D

ACUERDO DE LICENCIA DE CÓDIGO FUENTE DE AT&T

ast

ACUERDO DE CÓDIGO FUENTE
Versión 1.2D

POR FAVOR, LEA ESTE ACUERDO DETENIDAMENTE. Al acceder y utilizar el **Código Fuente**, acepta este Acuerdo en su totalidad y se compromete a utilizar el **Código Fuente** únicamente de acuerdo con los siguientes términos y condiciones. Si no desea quedar vinculado por estos términos y condiciones, no acceda ni utilice el **Código Fuente**.

1. TUS REPRESENTACIONES

   - Declara y garantiza que:

     a) Si eres una entidad, o un individuo que no es la persona que acepta este Acuerdo, la persona que acepta este Acuerdo en tu nombre es tu representante legalmente autorizado, debidamente autorizado para aceptar acuerdos de este tipo en tu nombre y obligarte a cumplir con sus disposiciones.

     b) Has leído y comprendido completamente este Acuerdo en su totalidad.

     c) Tus **Materiales de Construcción** son originales o no incluyen ningún **Software** obtenido bajo una licencia que entre en conflicto con las obligaciones contenidas en este Acuerdo.

     d) Según tu leal saber y entender, tus **Materiales de Construcción** no infringen ni apropian indebidamente los derechos de ninguna persona o entidad. Monitorizarás regularmente el Sitio web en busca de cualquier aviso.

2. DEFINICIONES E INTERPRETACIÓN

a) A los efectos de este Acuerdo, ciertos términos se han definido a continuación y en otros lugares de este Acuerdo para abarcar significados que pueden diferir o agregarse a la connotación normal de la palabra definida.

1) **«Código Adicional»** significa **Software** en forma de código fuente que no contiene ningún

   - **Código Fuente**, o
   - trabajo derivado (este término tiene el mismo significado en este Acuerdo que en la Ley de Derechos de Autor de EE. UU.) del **Código Fuente**.

2) **«Reclamaciones de Patente de AT&T** significa aquellas reclamaciones de patentes (i) propiedad de AT&T y (ii) licenciables sin restricción u obligación, que, sin una licencia, son necesaria e inevitablemente infringidas por el uso de la funcionalidad del **Código Fuente**.

3) **«Materiales de Construcción»** significa, en referencia a un **Producto Derivado**, el **Parche** y el **Código Adicional**, si los hay, utilizados en la preparación de dicho **Producto Derivado**, junto con instrucciones escritas que describan, de manera razonable detallada, dicha preparación.

4) **«Cápsula»** significa un archivo informático que contiene exactamente el mismo contenido que un archivo informático descargado del **Sitio web**.

5) **«Producto Derivado»** significa un **Producto de Software** que es una obra derivada del **Código Fuente**.

6) **«IPR»** significa todos los derechos protegibles bajo la ley de propiedad intelectual en cualquier lugar del mundo, incluidos los derechos protegibles bajo las leyes de patentes, derechos de autor y secretos comerciales, pero no los derechos de marca.

7) **«Parche»** significa **Software** para cambiar toda o cualquier parte del **Código Fuente**.

8) **«Aviso Propietario»** significa la siguiente declaración:

   «Este producto contiene ciertos códigos de software u otra información («Software de AT&T») propietarios de AT&T Corp. («AT&T»). El Software de AT&T se proporciona tal cual. USTED ACEPTA TODA LA

RESPONSABILIDAD Y RIESGO POR EL USO DEL SOFTWARE DE AT&T. AT&T NO REALIZA, Y RENUNCIA EXPRESAMENTE, CUALQUIER GARANTÍA EXPRESA O IMPLÍCITA DE NINGÚN TIPO, INCLUIDAS, SIN LIMITACIÓN, LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD O ADECUACIÓN PARA UN PROPÓSITO PARTICULAR, LAS GARANTÍAS DE TÍTULO O NO INFRACCIÓN DE NINGÚN DERECHO DE PROPIEDAD INTELECTUAL, CUALQUIER GARANTÍA QUE SURJA POR EL USO DEL COMERCIO, CURSO DE NEGOCIACIÓN O DESEMPEÑO, O CUALQUIER GARANTÍA DE QUE EL SOFTWARE DE AT&T ES «LIBRE DE ERRORES» O CUMPLIRÁ CON SUS REQUISITOS.

A menos que acepte una licencia para utilizar el Software de AT&T, no debe descompilar, desensamblar o realizar ingeniería inversa de este producto para averiguar el código fuente de ningún Software de AT&T.

© AT&T Corp. Todos los derechos reservados. AT&T es una marca registrada de AT&T Corp.»

9) **«Software»** significa, según lo que pueda requerir el contexto, instrucciones de código fuente u objeto para controlar el funcionamiento de una unidad central de procesamiento o computadora, y archivos informáticos que contienen datos o texto.

10) **«Producto de Software»** significa una colección de archivos informáticos que contienen **Software** solo en forma de código objeto, que, tomados en conjunto, comprenden razonablemente un producto, independientemente de si dicho producto está destinado para uso interno o explotación comercial. Un solo archivo informático puede constituir un **Producto de Software**.

11) **«Código Fuente»** significa el **Software** contenido en forma comprimida en la **Cápsula**.

12) **«Sitio web»** significa el sitio web de Internet con la URL http://www.research.att.com/sw/download/. AT&T puede **cambiar** el contenido o la URL del **Sitio web**, o eliminarlo por completo de Internet.

b) A modo de aclaración solamente, los términos **Cápsula, Aviso Propietario** y **Código Fuente**, cuando se utilizan en este Acuerdo, significarán los mate-

riales e información definidos por dichos términos sin ningún cambio, mejora, enmienda, alteración o modificación (colectivamente, **«cambio»**).

3. CONCESIÓN DE DERECHOS

a) Sujeto a reclamaciones de propiedad intelectual de terceros, si las hay, y a los términos y condiciones de este Acuerdo, AT&T te otorga bajo:

1) Las **Reclamaciones de Patente** de AT&T y los derechos de autor de AT&T en el **Código Fuente**, una licencia no exclusiva y totalmente pagada para:

a′ Reproducir y distribuir la **Cápsula**.

b′ Mostrar, ejecutar, utilizar y compilar el **Código Fuente** y ejecutar el **Software** binario resultante en una computadora.

c′ Preparar un **Producto Derivado** únicamente compilando **Código Adicional**, si lo hay, junto con el código resultante de operar un **Parche** en el **Código Fuente**.

d′ Ejecutar en una computadora y distribuir a otros **Productos Derivados**, excepto que, con respecto a las **Reclamaciones de Patente de AT&T**, los derechos de licencia otorgados en los incisos (iii) y (iv) anteriores solo se extenderán y limitarán a esa parte de un **Producto Derivado** que es un **Software** compilado a partir de alguna parte del **Código Fuente**; y,

2) Los derechos de autor de AT&T en el **Código Fuente**, una licencia no exclusiva y totalmente pagada para preparar y distribuir **Parches** para el **Código Fuente**.

b) Sujeto a los términos y condiciones de este Acuerdo, puedes crear un hipervínculo entre un sitio web de Internet de tu propiedad y controlado por ti y el **Sitio web**, que describa de manera justa y de buena fe dónde se pueden obtener la **Cápsula** y el **Código Fuente**, siempre que no enmarques el **Sitio web** ni des la falsa impresión de que AT&T está de alguna manera asociado, respalda o patrocina tu sitio web. Cualquier buena voluntad asociada con dicho hipervínculo será para el único beneficio de AT&T. Además de la creación de dicho hipervínculo, nada en este Acuerdo se interpretará como que te confiere el derecho de

usar cualquier referencia a AT&T, sus nombres comerciales, marcas comerciales, marcas de servicio o cualquier otro indicio de origen propiedad de AT&T, o indicar que tus productos o servicios están de alguna manera patrocinados, aprobados o respaldados por, o afiliados con, AT&T.

c) Excepto según se establece expresamente en la Sección 3.1 anterior, no se otorgan ni otros derechos ni licencias bajo ninguna **IPR** de AT&T ni, por implicación, impedimento o de otra manera, se confieren. A modo de ejemplo únicamente, no se otorgan ni se confieren derechos ni licencias bajo ninguna de las patentes de AT&T ni, por implicación, impedimento o de otra manera, con respecto a ninguna parte de un **Producto Derivado** que no sea un **Software** compilado a partir de alguna parte, sin **cambios**, del **Código Fuente**.

4. TUS OBLIGACIONES

a) Si distribuyes **Materiales de Construcción** (incluido si estás obligado a hacerlo en virtud de este Acuerdo), debes asegurarte de que el destinatario celebre y acepte debidamente un acuerdo contigo que incluya los términos mínimos establecidos en el Apéndice A (http://www.research.att.com/sw/license/ast-terms.html) (completado para indicar que eres el LICENCIANTE) y ninguna otra disposición que, en opinión de AT&T, entre en conflicto con tus obligaciones según, o el propósito de, este Acuerdo. El acuerdo requerido en virtud de esta Sección 4.1 puede estar en forma electrónica y puede distribuirse con los **Materiales de Construcción** de una manera tal que el destinatario acepte el acuerdo mediante el uso o la instalación de los **Materiales de Construcción**. Si algún **Código Adicional** contenido en tus **Materiales de Construcción** incluye **Software** que obtuviste bajo licencia, el acuerdo también incluirá detalles completos sobre la licencia y cualquier restricción u obligación asociada con dicho **Software**.

b) Si preparas un Parche que distribuyes a cualquier otra persona, debes:

1) Ponerte en contacto con AT&T, según se pueda proporcionar en el **Sitio web** o en un archivo de texto incluido con el **Código Fuente**, y describir para AT&T dicho **Parche** y proporcionar a AT&T una copia de dicho **Parche** según lo indique AT&T; o,

2) Donde hagas tu **Parche** generalmente disponible en tu sitio web de Internet,

debes proporcionar a AT&T la URL de tu sitio web y, por la presente, otorgar a AT&T un derecho no exclusivo y totalmente pagado para crear un hipervínculo entre tu sitio web y una página asociada al **Sitio web**.

c) Si preparas un **Producto Derivado**, dicho producto debe mostrar de manera conspicua a los usuarios, y cualquier documentación y acuerdo de licencia correspondiente debe incluir como disposición el **Aviso Propietario**.

5. TU CONCESIÓN DE DERECHOS A AT&T

a) Concedes a AT&T, bajo cualquier **propiedad intelectual de tu propiedad** o con licencia tuya que de alguna manera esté relacionada con tus **Parches**, una licencia no exclusiva, perpetua, mundial, totalmente pagada, sin restricciones, irrevocable, junto con el derecho de sublicenciar a otros, para (a) fabricar, hacer fabricar, usar, ofrecer en venta, vender e importar cualquier producto, servicio o cualquier combinación de productos o servicios, y (b) reproducir, distribuir, preparar trabajos derivados basados en, realizar, mostrar y transmitir tus Parches en cualquier medio, ya sea conocido ahora o desarrollado en el futuro.

6. CLAUSULA «TAL COMO ESTÁ» / LIMITACIÓN DE RESPONSABILIDAD

a) El **Código Fuente** y la **Cápsula** se proporcionan «TAL COMO ESTÁ». ASUMES LA RESPONSABILIDAD Y RIESGO TOTAL POR SU USO, INCLUYENDO EL RIESGO DE DEFECTOS O INEXACTITUDES. AT&T NO REALIZA, Y RENUNCIA EXPRESAMENTE A, CUALQUIER GARANTÍA EXPRESA O IMPLÍCITA DE NINGÚN TIPO, INCLUYENDO, SIN LIMITACIÓN, LAS GARANTÍAS IMPLÍCITAS DE COMERCIABILIDAD O ADECUACIÓN PARA UN PROPÓSITO PARTICULAR, GARANTÍAS DE TÍTULO O NO INFRACCIÓN DE CUALQUIER DERECHO DE PROPIEDAD INTELECTUAL O MARCA, CUALQUIER GARANTÍA QUE SURJA DEL USO DEL COMERCIO, CURSO DE TRATAMIENTO O DESEMPEÑO, O CUALQUIER GARANTÍA DE QUE EL CÓDIGO FUENTE O LA CÁPSULA SON «LIBRES DE ERRORES» O CUMPLIRÁN CON TUS REQUISITOS.

b) EN NINGÚN CASO AT&T SERÁ RESPONSABLE DE (a) DAÑOS INCIDENTALES, CONSECUENTES O INDIRECTOS (INCLUYENDO, SIN LIMITACIÓN, DAÑOS POR PÉRDIDA DE BENEFICIOS, INTERRUPCIÓN DEL NEGOCIO, PÉRDIDA DE PROGRAMAS O INFORMACIÓN, Y SIMI-

LARES) DERIVADOS DEL USO O LA INCAPACIDAD DE USAR EL **CÓDIGO FUENTE** O **LA CÁPSULA**, INCLUSO SI SE HA INFORMADO A AT&T O A CUALQUIERA DE SUS REPRESENTANTES AUTORIZADOS DE LA POSIBILIDAD DE TALES DAÑOS, (b) CUALQUIER RECLAMO ATRIBUIBLE A ERRORES, OMISIONES U OTRAS INEXACTITUDES EN EL **CÓDIGO FUENTE** O **LA CÁPSULA**, O (c) CUALQUIER RECLAMO DE TERCEROS.

c) DEBIDO A QUE ALGUNOS ESTADOS NO PERMITEN LA EXCLUSIÓN O LIMITACIÓN DE RESPONSABILIDAD POR DAÑOS INCIDENTALES O CONSECUENTES, ES POSIBLE QUE LAS LIMITACIONES ANTERIORES NO SE APLIQUEN A TI. EN CASO DE QUE LA LEGISLACIÓN APLICABLE NO PERMITA LA EXCLUSIÓN O LIMITACIÓN TOTAL DE RESPONSABILIDAD DE RECLAMACIONES Y DAÑOS SEGÚN LO ESTABLECIDO EN ESTE ACUERDO, LA RESPONSABILIDAD DE AT&T SE LIMITA EN LA MAYOR MEDIDA PERMITIDA POR LA LEY.

7. INDEMNIZACIÓN

a) Indemnizarás y eximirás de responsabilidad a ATi&T, sus afiliados y representantes autorizados contra cualquier reclamo, demanda o procedimiento presentado o iniciado por cualquier tercero y derivado de, o relacionado con, tu uso del **Código Fuente**. Esta obligación incluirá indemnizar contra todos los daños, pérdidas, costos y gastos (incluidos los honorarios de abogados) incurridos por AT&T, sus afiliados y representantes autorizados como resultado de tales reclamaciones, demandas o procedimientos, incluidos los costos o gastos incurridos en la defensa contra tales reclamaciones, demandas o procedimientos.

8. GENERAL

a) No presentarás contra AT&T, sus afiliados o representantes autorizados ningún reclamo por infracción o apropiación indebida de cualquier **propiedad intelectual** o derechos de marca de alguna manera relacionados con el **Código Fuente**, incluidos aquellos relacionados con cualquier **Parche**.

b) En caso de que alguna disposición de este Acuerdo se considere ilegal o inaplicable, AT&T puede, pero no está obligada a, publicar en el **sitio web** una nueva versión de este Acuerdo que, en opinión de AT&T, preserve razonablemente la

intención de este Acuerdo.

c) Tus derechos y licencia (pero no tus obligaciones) bajo este Acuerdo se terminarán automáticamente en caso de que (a) se publique en el **sitio web** un aviso de una reclamación no frívola de un tercero relacionada con el **Código Fuente** o la **Cápsula**, (b) tengas conocimiento de dicha reclamación, (c) alguna de tus representaciones o garantías en el Artículo 1.0 o la Sección 8.4 sean falsas o inexactas, (d) excedas los derechos y la licencia otorgados o (e) no cumplas completamente con cualquier disposición de este Acuerdo. Nada en esta disposición se interpretará para restringirte, a tu opción y sujeto a la ley aplicable, de reemplazar la parte del **Código Fuente** que sea objeto de una reclamación de un tercero con código no infractor o de negociar de manera independiente los derechos necesarios con el tercero.

d) Reconoces que el **Código Fuente** y la **Cápsula** pueden estar sujetos a las leyes y regulaciones de exportación de EE. UU. y, en consecuencia, garantizas a AT&T que no violarás, directa o indirectamente, ninguna ley o regulación de exportación de los EE. UU. aplicable.

e) Sin limitar ninguno de los derechos de AT&T en virtud de este Acuerdo o por ley o en equidad, ni ampliar de otra manera el alcance de la licencia y los derechos otorgados en virtud del presente, si no cumples con alguna de tus obligaciones bajo este Acuerdo con respecto a cualquiera de tus **Parches** o **Productos Derivados**, o si realizas cualquier acto que exceda el alcance de la licencia y los derechos otorgados en este documento, entonces dichos **Parches, Productos Derivados** y actos no están licenciados ni autorizados en virtud de este Acuerdo y dicho incumplimiento también se considerará una violación de este Acuerdo. Además de cualquier otro remedio disponible por cualquier violación de tus obligaciones bajo este Acuerdo, AT&T tendrá derecho a una orden judicial que te obligue a cumplir con dichas obligaciones.

f) Este Acuerdo se regirá e interpretará de acuerdo con las leyes del Estado de Nueva York, EE. UU., sin tener en cuenta sus reglas de conflictos de leyes. Este Acuerdo será interpretado de manera justa de acuerdo con sus términos y sin ninguna interpretación estricta a favor o en contra de AT&T o de ti. Cualquier demanda o procedimiento que presentes en relación con este Acuerdo deberá presentarse y tramitarse únicamente en Nueva York, Nueva York, EE. UU.